# NAVAL POSTGRADUATE SCHOOL
## Monterey, California






# THESIS

**NPSNET: PHYSICALLY BASED, AUTONOMOUS, NAVAL SURFACE AGENTS**

by

John Henry Hearne, Jr.

September 1993

Thesis Advisor: Dr. David R. Pratt
Co-Advisor: Dr. Sehung Kwak

# REPORT DOCUMENTATION PAGE

*Form Approved*
*OMB No. 0704-0188*



| 1. AGENCY USE ONLY (Leave Blank) | 2. REPORT DATE | 3. REPORT TYPE AND DATES COVERED |
|---|---|---|
| | September 1993 | Master's Thesis |

**4. TITLE AND SUBTITLE**
NPSNET: Physically Based, Autonomous, Naval Surface Agents

**5. FUNDING NUMBERS**

**6. AUTHOR(S)**
Hearne Jr., John Henry

**7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)**
Naval Postgraduate School
Monterey, CA 93943-5000

**8. PERFORMING ORGANIZATION REPORT NUMBER**

**9. SPONSORING/ MONITORING AGENCY NAME(S) AND ADDRESS(ES)**

**10. SPONSORING/ MONITORING AGENCY REPORT NUMBER**

**11. SUPPLEMENTARY NOTES**
The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the United States Government.

**12a. DISTRIBUTION / AVAILABILITY STATEMENT**
Approved for public release;
distribution is unlimited

**12b. DISTRIBUTION CODE**

**13. ABSTRACT** *(Maximum 200 words)*

The Computer Science Department at the Naval Postgraduate School in Monterey, California has developed a low-cost real-time interactive simulation system using the Distributed Interactive Simulation (DIS) Protocol, known as NPSNET, that works on commercially available Silicon Graphics IRIS workstations.

In NPSNET, vehicular movement is determined by either a script or by control through input devices. A few vehicles have a reactive intelligent capability, but none possess the ability to cooperate and interact with one another. Additionally, there are no ships incorporated into NPSNET. Therefore, the problem addressed by this thesis is to add intelligent, autonomous movement to physically based vehicles in NPSNET.

The approach is to use an expert systems tool, CLIPS, to simulate naval surface units, modeled using

| 14. SUBJECT TERMS | 15. NUMBER OF PAGES |
|---|---|
| Graphics, CLIPS, Autonomous, Expert Systems, Physically Based Modeling, Ships | 81 |
| | 16. PRICE CODE |

| 17. SECURITY CLASSIFICATION OF REPORT | 18. SECURITY CLASSIFICATION OF THIS PAGE | 19. SECURITY CLASSIFICATION OF ABSTRACT | 20. LIMITATION OF ABSTRACT |
|---|---|---|---|
| UNCLASSIFIED | UNCLASSIFIED | UNCLASSIFIED | UL |

[ item 13 ] Continued: computer graphics, for evaluating the effectiveness of this control method. The rules were developed and debugged on a test platform and then networked to NPSNET. Under the NPSNET harness, the autonomous forces are handled separately from the main program, thus reducing processor time and allowing for more complex environments.

There are several noteworthy accomplishments resulting from this work. First is the ability to interface graphics C functions with CLIPS, actually invoking and controlling graphics programs from the CLIPS prompt. Second is the development of an autonomous agents test bed. The rules from this test bed are then incorporated into the NPSNET autonomous agent control program. Third is the development of intelligent physically based ships which the ability to maneuver to avoid collisions with static and non-static objects. Fourth, the foundation for future work on rule based simulation is in place. Finally, there are autonomous, physically based naval surface forces that can operate over a DIS network realistically, in real-time.

Approved for public release; distribution is unlimited

# NPSNET: PHYSICALLY BASED, AUTONOMOUS, NAVAL SURFACE AGENTS

by

*John H. Hearne, Jr.*
*Lieutenant, United States Navy*
*B.A. Mathematics, University of North Carolina, 1985*

Submitted in partial fulfillment of the
requirements for the degree of

**MASTER OF COMPUTER SCIENCE**

from the

**NAVAL POSTGRADUATE SCHOOL**
September 1993



Author: John H. Hearne



Approved By:

Dr. David R. Pratt, Thesis Advisor

Dr. Sehung Kwak, Thesis Co-Advisor

Dr. Ted G. Lewis, Chairman,
Department of Computer Science

# ABSTRACT

The Computer Science Department at the Naval Postgraduate School in Monterey, California has developed a low-cost real-time interactive simulation system using the Distributed Interactive Simulation (DIS) Protocol, known as NPSNET, that works on commercially available Silicon Graphics IRIS workstations.

In NPSNET, vehicular movement is determined by either a script or by control through input devices. A few vehicles have a reactive intelligent capability, but none possess the ability to cooperate and interact with one another. Additionally, there are no ships incorporated into NPSNET. Therefore, the problem addressed by this thesis is to add intelligent, autonomous movement to physically based vehicles in NPSNET.

The approach is to use an expert systems tool, CLIPS, to simulate naval surface units, modeled using computer graphics, for evaluating the effectiveness of this control method. The rules were developed and debugged on a test platform and then networked to NPSNET. Under the NPSNET harness, the autonomous forces are handled separately from the main program, thus reducing processor time and allowing for more complex environments.

There are several noteworthy accomplishments resulting from this work. First is the ability to interface graphics C functions with CLIPS, actually invoking and controlling graphics programs from the CLIPS prompt. Second is the development of an autonomous agents test bed. The rules from this test bed are then incorporated into the NPSNET autonomous agent control program. Third is the development of intelligent physically based ships which the ability to maneuver to avoid collisions with static and non-static objects. Fourth, the foundation for future work on rule based simulation is in place. Finally, there are autonomous, physically based naval surface forces that can operate over a DIS network realistically, in real-time.

# TABLE OF CONTENTS

# LIST OF FIGURES

# I. INTRODUCTION

Today's military faces challenges unlike any other in the post World War II era. Since the end of the Cold War, many elected officials on Capitol Hill are attempting to cash in on the "peace dividend". Daily the news from Washington is about the shrinking military budget, downing sizing of forces and restructuring or eliminating military bases. However, the military's requirement to maintain full combat readiness has not changed. Neither have the deployments and other commitments. Today's military is participating around the globe, both with United Nations forces and unilaterally. In order to maintain training levels at the required highest level, alternatives to costly full scale, integrated maneuvers must be developed. With the rapid ascent of computer technology and the corresponding decline in cost, state of the art interactive simulation systems are proving to be an effective yet economical alternative. Today's fighting men and women can participate in realistic battlefield simulations safely and cost effectively using this technology. By networking the three dimensional virtual worlds battlespace simulators, many players have the opportunity to develop and hone their warfighting skills at a fraction of the cost necessary to conduct live training exercises.

The Computer Science Department at the Naval Postgraduate School in Monterey, California has developed a low cost battlespace simulation system, known as NPSNET [Zyda92]. NPSNET is designed to work on commercial off-the-shelf Silicon Graphics 4D workstations. Initially it was geared towards ground forces and land based conflicts. Recently, more work has been directed towards constructing a naval component for NPSNET, therefore attaining a more realistic joint services simulator. This thesis is a step in that direction by incorporating naval surface units into NPSNET.

The primary purpose of this research is to develop a proof of concept model for the interaction of an expert system and graphics. In the previous versions of NPSNET, graphical vehicles were controlled by prewritten script or player interaction. Vehicles in this thesis work are autonomous, they exist or are capable of existing independently. No

prescripted movement is used, they react to environmental conditions. This thesis shows how an expert system shell can be the controlling program in a graphics world. The expert system makes the vehicles "smart", able to react to any situation, not just follow a script. This type of smart battlespace simulator will allow for much more realism in training for tomorrow's armed forces.

Chapter II provides an overview of NPSNET. Chapter III explains what an expert system is and why they are important to this domain. Chapter IV gives an overview and the goals of this work. Chapter V examines the graphical models and their dynamic physically based movement. Chapter VI discusses the expert system implementation and the various sets of rules developed. Chapter VII discusses incorporating the autonomous players into NPSNET IV. Chapter VIII is the summary of conclusions and further work.

Appendix A is the makefile used in the autonomous agents test bed. Appendix B (CLIPS) and Appendix C (graphics) contain the code for a sample program which illustrates how CLIPS can serve as an upper level decision maker for a graphical program. Their relationship is discussed in detail throughout this thesis.

# II. OVERVIEW OF NPSNET

NPSNET is a real-time, low-cost, three dimensional visual simulation system, developed by the researchers in the Computer Science Department at the Naval Postgraduate School in Monterey, California [Zyda92]. NPSNET uses off-the-shelf graphics workstations from Silicon Graphics, Inc., instead of contractor produced machines. The computers are the same as those used to produced the realistic visual effects in the movies "Terminator II" and "Jurassic Park". In NPSNET, a participant may control any of 500 active vehicles using a six degree of freedom Spaceball or throttles and joysticks and observe the detailed features of the environment such as the roads, buildings, lakes and mountains. Other vehicles are controlled by participants at various workstations, either in the lab or over the Distributed Interactive Simulation (DIS) network, or by expert systems or by NPSNET itself. Communications between workstations in the laboratory is accomplished by broadcasting locally designed packets on an Ethernet network. NPSNET is also equipped to transmit in the DIS protocol, which allows communication with players on a national level.

Objects are modeled using an object description language developed at Naval Postgraduate School. The NPS Object File Format (NPSOFF) is an ASCII formatted file that incorporates many SGI graphics library (GL) calls. NPSOFF relieves the programmer of the burden of designing and rendering the object, thus allowing concentrated effort on how the object should be used. Future improvements include development of the Graphics Description Language, an object oriented method of encapsulating the model information, using the programming language C++.

Because NPSNET was originally implemented as a land based battlefield simulator, there are minimal naval elements involved. Inclusion of this work and other recent thesis work with naval components, will enable NPSNET to provide a more realistic, joint approach in the battlespace simulator.

# III. EXPERT SYSTEMS OVERVIEW

An expert system, as defined by Professor E. Feigenbaum of Stanford University, a pioneer in the Artificial Intelligence field, is:

> an intelligent computer program that uses knowledge and inference procedures to solve problems that are difficult enough to require significant human expertise for their solution. The knowledge necessary to perform at such a level, plus the inference procedures used, can be thought of as a model of the expertise of the best practitioners of the field. [Walk90]

The knowledge engineer enters the expert's knowledge into the expert system. When fully implemented, the expert systems' knowledge base is greater than the sum of the individual expert's knowledge. The expert system can communicate with the client, reason within the knowledge base and then give client advice and even explain the reasoning [Sieg86].

## A. REPRESENTING KNOWLEDGE

Knowledge can best be described in a hierarchal relationship with data and information. Data or a fact is the most basic element. Facts are indisputable, such as *Matthew is two years old* or *Kristi has blue eyes*. Information is a collection of facts. The facts are combined, summarized, collated, compared, classified, associated, or otherwise processed to make human decision-making easier. The highest level is knowledge, the basis for human decision making. It takes knowledge to interpret the information and determine the most correct response [Sieg86].

Representing knowledge is a non-trivial task. Although instinctive, knowledge is a vague term and therefore difficult to pinpoint. Research in knowledge representation is being conducted on various descriptive, procedural and mathematical techniques. The three most common representations are semantic networks, frames, and rules [Sieg86].

## 1. Semantic Networks

A semantic network shows relationships between different entities. Mathematicians would classify it as a *labeled directed graph* [Rowe88]. Figure 1 illustrates the relational behavior. The objects in the illustration are connected with a line with the

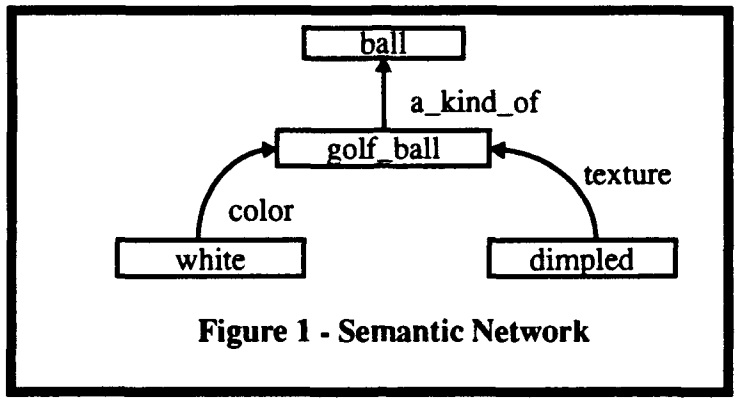



**Figure 1 - Semantic Network**

relational description written beside the line, i.e. the golf_ball is the color white and a golf_ball is a_kind_of ball. Semantic networks can be used to describe systems and problems, and the relations can be manipulated by computer.

## 2. Frames

The frame representation of knowledge is based upon Marvin Minsky's theory about how humans think. "A frame is a data structure for representing a stereotyped situation....Attached to each frame are several kinds of information. Some is about what one can expect to happen next. Some is about what to do if these expectations are not confirmed" [Sieg86]. The data from the example semantic network is represented by frames in Figure 2. Two advantages of frames over semantic networks are that frames may be used for partitioning a complex domain and storing procedures in addition to descriptive data.

## 3. Rules

Although semantic networks and frames are simple to understand and easy to illustrate, one of the most popular methods of representing knowledge is using a rule based

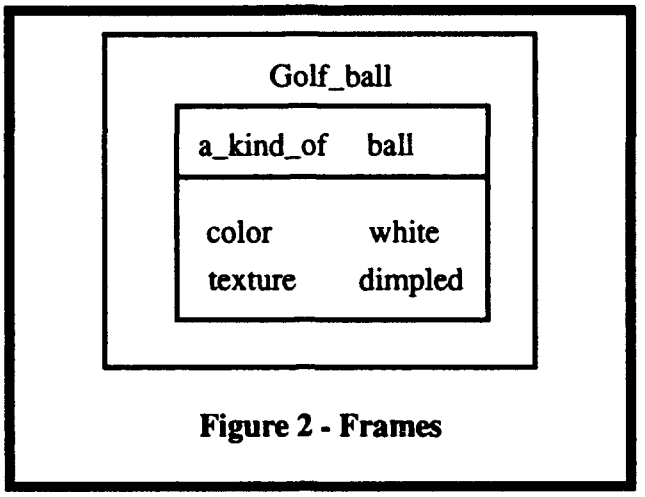



**Figure 2 - Frames**

system. Rules have several advantages as both [Sieg86] and [Walk90] point out. First, rules are simple. They are easy to express, understand and work with. Rule expressions are interpreted as "if-then" statements. Secondly, rules are modular. Each rule expresses a separate thought and can be changed or modified without affecting other rules. Also rules will be the appropriate size. They are broken down to the simplest terms and related by the conclusion of one statement being the conditional of another rule. Rules are both procedural and descriptive. Rule based languages can provide explanations of actions. They can be configured to explain which rules it used and why.

In human reasoning, we often follow certain guidelines called *rules-of-thumb. An apple a day keeps the doctor away* is a rule-of-thumb. Rule based expert systems are well suited to convert rules-of-thumb into languages machines understand. A rule consists of two parts: the conditional (IF) part and the conclusion (THEN) part. If a certain condition exists, then the rule is executed and a known result will occur. A frequently used golf example is *if you lift your head then you will hit a bad shot.*

The *inference engine* is the reasoning machine. It sequences through the rules of the knowledge base, queries the user for input and provides answers based on the user's information and the rules. An inference engine for a rule based system is referred to as a *rule interpreter* [Walk90]. Siegel puts it best when describing the rule based expert system, when he says:

rules are modular, pithy chunks of knowledge, which can be replaced or modified without affecting other rules. Rules are the basic building blocks of the knowledge base which stores your expertise. Expert systems built around such knowledge bases are flexible, adaptable to changing conditions and able to handle complex problems. [Sieg86]

## B. CHAINING

Machine reasoning, or chaining, is the path the computer traces as it sequences through the knowledge base. If the machine reasons from known facts forward to goals, the process is called *forward chaining*. Forward chaining is also called *data-directed computation* or *modus ponens* reasoning because it depends on facts to reach a conclusion. If a machine reasons backwards from goals to facts, the process is called *backward chaining* or *goal-directed reasoning* [Rowe88]. Goal directed reasoning takes an attained goal and searches for the facts that led to that goal.

An example of forward chaining is driving to The Lodge at Pebble Beach. The desired result is to arrive at The Lodge. The tourist knows that the rental car came with keys and that the keys will start the car. He knows that he can drive the car along Highway One to the 17 Mile Drive gate and pay the $6.00 entry fee. The tourist knows if he stays on 17 Mile Drive, he will arrive on the Pebble Beach grounds. Upon finding a parking place, the tourist can walk to the Lodge and the goal is attained. The tourist used the facts (car, roads, entry fee, etc.) and reasoned his way forward until arriving at his goal.

An example of backward chaining is determining the reason(s) for a heart attack. The doctor knows that the patient has attained the goal (the heart attack). Now the facts which led to the goal must be ascertained. The doctor checks the medical records to determine if the cholesterol level was excessive and if family history indicated previous heart disease. She then checks to see if the patient were a smoker, had a sedentary life-style or an unhealthy diet. The doctor backtracks by checking all information about the patient until one or more of the facts determines the reason the goal was attained.

When developing an expert system, one must determine which type of reasoning is most appropriate for their application. Each problem most likely will present itself as either a data-driven problem or goal-driven problem. The developer must then choose an inference engine which optimizes the dominant chaining method.

## C. OTHER FEATURES

There are three other important features of an expert system, system-client communications, uncertainty, and explanation. In order for an expert system to effectively provide assistance, a mechanism must be installed to ensure full intercommunications between the expert system (server) and the user (client). This communication link is essential so that the client's requirements are fully met and so that the expert system provides the most correct response for the given input. Querying the user is not the only means of information gathering available to the system. Depending upon configuration, the expert system may have access to software databases, spreadsheets and statistical packages.

Rules entered into the expert system are empirical rules, they work based on the expert's experience. Different certainty factors may be placed on the rules and the expert system renders advice based on the confidence it has on the rules.

Expert systems also have the ability to trace their reasoning path when determining results. This is especially helpful in tracing how and why the system produced a certain result.

## D. CLIPS

The C Language Integrated Production System (CLIPS) is a forward chaining inference engine (expert system tool) developed by NASA-Lyndon B. Johnson Space Center. CLIPS is designed to facilitate the development of software to model human knowledge or expertise. There are two versions, one written in the C programming language and one written in Ada. [Giar91]

There are three ways to represent knowledge in CLIPS. The first is rules, containing heuristic knowledge based on experience. Second is functions which allows procedural

knowledge and third is object-oriented programming. CLIPS provides a completely supported Object Oriented Programming environment.

CLIPS is very flexible for the knowledge engineer in that it provides two means of interacting with a procedural language. One is that CLIPS can be called from a procedural language (C, Ada), perform its function and then return control to the calling program. Alternately, CLIPS may call functions from the procedural code. When the function completes its task, control is returned to CLIPS.

# IV. SYSTEM OVERVIEW

The primary purpose of this research is to develop a proof of concept model for the interaction of an expert system and graphics. In the previous versions of NPSNET, graphical vehicles were controlled by prewritten script or player interaction. Vehicles in this thesis work are autonomous, they exist and act independently of the user. No prescripted movement is used, they react to environmental conditions. This thesis shows how an expert system shell can be the controlling program in a graphics world. The expert system makes the vehicles "smart". The surface vessels are modeled and rendered using dynamic, physically based models, to improve the realism. The effect is more that of a naval surface simulator rather than that of a video game. This type of smart battlespace simulator could allow for much more realism in training for tomorrow's armed forces. Thus the major goal of this research is to develop a proof of concept model, designed to demonstrate realistic, real-time, autonomous, physically based models capable of reacting correctly to ever changing environment.

## A. SHIP DYNAMIC MOVEMENT OVERVIEW

Surface ship movement at sea is complex, with many factors affecting its motion. There are ship related factors such as speed, heading, turning rate and rudder angles and environmental factors such as wind, currents and tides. Models in this thesis incorporate the ship related factors into its movement calculations. The two primary areas addressed are turning dynamics and ship propulsion dynamics.

## B. GRAPHICS SUMMARY

This thesis is not concerned with complex model building or constructing programs which display spectacular scenes. It is concerned with displaying models which accurately simulate the actions of the vessels they represent. Thus when the program is running, the viewer will see just three ships sailing in a lake. The rendered images are not the important issues, the reason behind their movement is of the paramount importance.

The models used were received from ARPA's SIMNET program and are formatted in the NPSOFF format. The actual rendering of the models is accomplished using standard graphics library calls on the Silicon Graphics machines. The important distinction between this work and other previous graphics programs at NPS, is that C is no longer the high level manager of the graphics program, that responsibility lies with CLIPS, the expert system shell. Therefore, the program is started from the CLIPS prompt and is controlled by C.

## C. EXPERT SYSTEM OVERVIEW

Initially the goal was to prove that graphics could successfully be controlled by an expert system. When this was accomplished, the goal was to build a knowledge base for surface ships. The desire was for the virtual world to handle situations real naval ships face during at-sea periods. The first and foremost goal was to ensure safe passage of the ships. To accomplish this goal, rules have to handle situations where ships are sailing in close proximity to land and must maneuver to avoid running aground. The capability to safely sail the open ocean requires rules which consider the ship's actions and that of others around. The ability to maneuver if necessary to avoid collision is considered. Actual international maritime rules of the road are encoded, which adds to the realism of the ship's actions.

An interaction between a helicopter and the surface ships is represented in this world. The helicopter is able to fly around the world, launch and recover from two of the surface units. A virtual world with naval surface units would not be complete without the ability to shoot down an enemy plane or missile. Therefore, rules were developed which allow the combatant in the world to successfully engage a target.

# V. PHYSICALLY BASED, DYNAMIC MODELING OF SURFACE SHIPS

Previous research in vehicle control models had been conducted with aircraft [Cook92] and [Schm93], underwater vehicles [Zehn93] and land-based vehicles [Schm93]. There had been no previous real-time dynamic modeling of surface ships at the Naval Postgraduate School. Thus initial motion control of the surface models was done with basic graphics library calls or with various input devices. The rendered motion of the models appeared cartoonish at times. The need for realistic motion for simulation was evident and required physically based, dynamic modeling.

## A. BACKGROUND

To control the motion of a naval surface ship, the Officer of the Deck orders course or heading changes and speed changes. The heading orders are given to the helmsman, who turns the helm the appropriate amount, activating the rudder, causing the ship to turn. There are two ways to give the heading order, either with the amount and direction of rudder and the new course to steer or with direction and the new ordered heading. The latter is normally done only for a course change of no more than ten degrees. An example of the former is *Left 15 degrees rudder, steady course 270.* The engine orders to the lee helmsman include the direction of the desired ship's movement (ahead or astern), the desired revolutions of the main shaft or shafts and the ordered speed. An example is *All ahead two thirds, indicate 055 revolutions for ten knots.* The lee helmsman places the engine order telegraph in the ordered position and the engineers in main control manipulate the propulsion plant to obey the engine order from the Officer of the Deck.

In the initial development of motion control, ship motion was dictated by a script or by using different input devices, such as the keyboard and the dialbox. The scripted movement motion could be quasi-realistic, if care was taken when developing the code. However, the motion inputted from the keyboard or dialbox was almost always unrealistic and cartoonish because the model could behave in a manner which was physically

impossible. The need for dynamic models was obvious if this work were to be used for simulation purposes and not just as a video game.

To model surface motion for this study, a set of motion control parameters was developed which cause the surface models to move in response to external stimulus. The external stimulus, corresponding to the Officer of the Deck, is represented by the expert system, CLIPS.

The goal was for the motion of the ship to as realistic as possible. There are different ways to visually inspect the motion of the model to determine realism. When turning, the ship should display the effects advance and transfer? Advance is defined as the distance gained in the direction of the original course. Transfer is the distance gained in a direction perpendicular to that of the original course line from the time the rudder is put over until on a new course, Figure 3. [SWOS85] The ship should appear to slide through the turn, not just pivot immediately about a point and continue on the new course. The turning rate

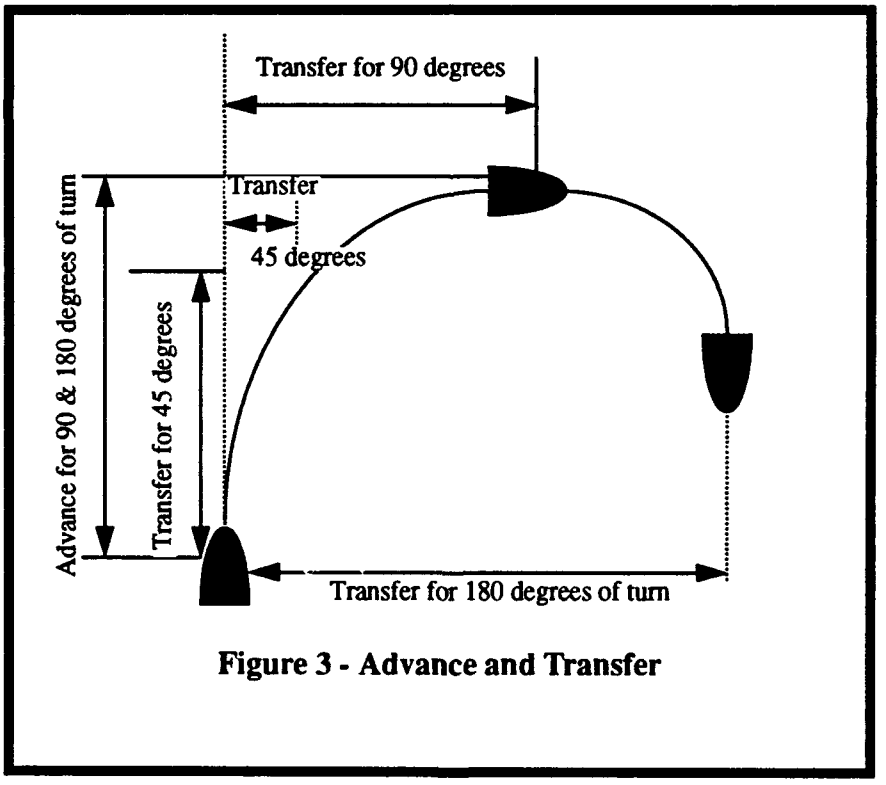



**Figure 3 - Advance and Transfer**

should be slow initially, gradually increasing, peaking, then decreasing until settling on the new course. The ship should accelerate or decelerate, not just immediately attained the new speed after the order is given. These are methods of displaying how a physically based ship can remove the cartoon effect and provide realism with the models.

## B. GENERAL MANEUVERS

The models and the environment had to be carefully scaled to provide for the realistic display. For example, if the speed of the ship is ten knots then the ship should travel 1000 yards in a three minute period. The scale of the ship model must correspond to the environment's units of measure. Therefore one "unit" along the x-axis is equal to one yard, and the ships are scaled accordingly.

Rendering the models is accomplished using the Euler angles and system calls to the graphics library to position and display the OFF objects. Euler angles are a common technique used for parameterization of orientation space where total rotations are described in terms of a sequence of rotations around the three axes. The order of rotations is critical. Different rotation combinations will result in a different final position, even when using the same values. [Watt92] However for the surface ships in this study, that problem is avoided since rotations are only considered around the y-axis, for heading change.

To maintain consistency in the speed of the motion from one graphics platform to another, all motion computations are based on the system real-time clock. Each time through the graphics loop, current system time is read and changes in ship's position is calculated as a function of the time difference from the previous time through the loop. Euler angles compute a system's derivatives at some time, $t_k$, and updating the data structures for some time, $t_{k+1}$, based on those derivatives and the time difference, $t_{k+1} - t_k$, is known as Euler's method [Barz92]. This is the least computational expensive numerical integration method and the least accurate. The accuracy provided by Euler angles is acceptable because the simulator is not intended to be a surface ship handling trainer. The

results are calculated quickly and with a sufficient degree of accuracy to provide realistic motion.

Assigning the motion to system time allows for consistent motion on any platform. There are several different models of Silicon Graphics machines in the lab, running on different CPU clock speeds. If motion were assigned to frame rate, the movement of the models would be different on each machine with a different clock speed. Synchronizing time to motion is also essential for network operations. When networking between the different machines, each must be able to move the model at the same speed during the dead reckoning time period between receipt of network updates. If the dead reckoning were calculated based on frame rate, the models would appear to jump around at the receiving station, because the dead reckoning position does not match that of the sending station.

## C. TURNING DYNAMICS

Using CLIPS, as the decision maker, emulates the Officer of the Deck. The OOD decides when to change course or speed. In this physically based representation of surface ships, the order to change course is simplified, only the desired new heading is given. The function, *turn_to_ordered_heading*, receives the ordered heading and turns the shortest path to the new heading. The amount of rudder is determined based on the amount of the course turn, but no more than 30 degrees of rudder is used. Thirty degrees of rudder is the maximum amount used in normal maneuvering situations. *Turn_to_ordered_heading* calculates when to shift the rudder, easing into the ordered heading. This function acts as the ship's helmsman. This method displays the advantage of using a high level expert system in conjunction with a high speed imperative language. CLIPS makes decisions based on information received, and orders an action. C receives the order and performs the calculations necessary to obey the order. This approach leaves the expert system free from concerning itself with small details, allowing it to maintain the overall, "big" picture, much as the Officer of the Deck is tasked.

## 1. Theory

To develop a correctly modeled, physically based ship, a basic understanding of ship dynamics is required. The background investigation was accomplished with the assistance of Professor Fotis Papoulias of the Mechanical Engineering Department at the Naval Postgraduate School. The information described is simplified for non-engineers and surface warfare officers, [PNA88] contains the in-depth details concerning ship turning and propulsion dynamics.

As a ship maneuvers through the water, there are many factors affecting the motion. Ship factors include, but are not limited to, the ship's heading and speed, the turning rate, the angle of the rudder, the responsiveness of the ship, and the size and strength of the rudder. Each of these factors are unique to each ship type. The handling characteristics of a cruiser will be significantly different than those of an aircraft carrier. Environmental and oceanographic factors are important considerations concerning the ship's maneuvering capabilities, but that area is not addressed in this work. There are start up projects in this area at NPS.

The fundamental turning dynamics equation is $\dot{r} = ar + b\delta$, where $a$ is the ship's turning responsiveness, $r$ is the turning rate (angular velocity), $b$ is related to the rudder's size and strength and $\delta$ is the rudder angle, Figure 4. The equation for determining rudder angle is $\delta = k_1\langle\psi - \psi_{com}\rangle + k_2 r$ where $k_1$ is the coefficient which relates the number of

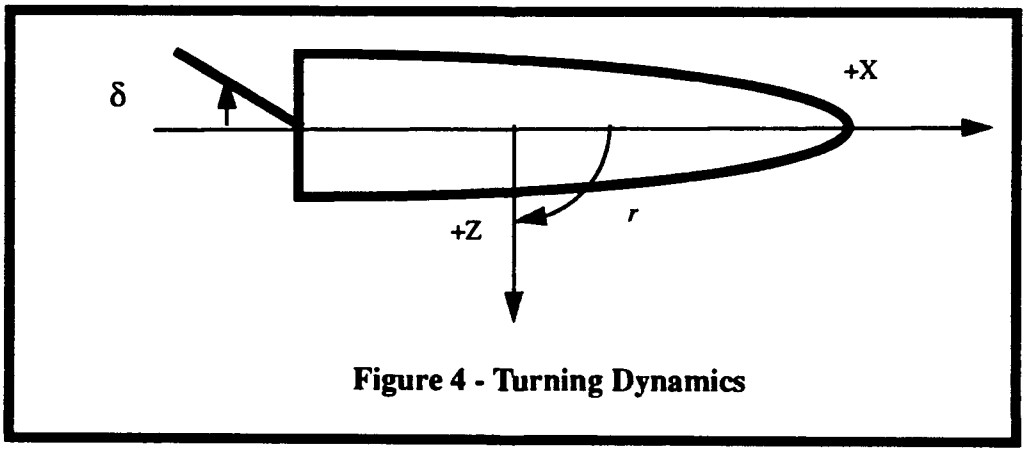



**Figure 4 - Turning Dynamics**

degrees of rudder per degree of heading difference, $\psi$ is the current ship's heading, $\psi_{com}$ is the ordered heading, $k_2$ is the rudder dampening variable and $r$ is the angular velocity.

These are the standard equations from [PNA88]. The equations are all generic, non ship type specific, but the required information is contained in those two basic formulae. The next step is converting the theoretical equations into computer code.

## 2. Implementation

In the implementation portion for turning dynamics, one major obstacle was confronted, we were unaware of the values for the variables necessary to solve the turning dynamics equations. The solution to this problem was two-fold, Professor Papoulias derived the relationships between the variables and the author used surface ship knowledge to closely approximate the other variables. For example, $k_1$ is the number of degrees of rudder to use per degree of course change. The Officer of the Deck uses as a rule of thumb, one degree of rudder for one degree of course change. The rudder coefficient, $k_1$,was implemented with a maximum limit of 30 degrees. Other variables such as the ship and rudder responsiveness and the rudder dampening coefficient were experimentally found based on their known relationships and observing the motion on the computer screen. The final values, when used to render the ship models, closely resembled the motion based upon the author's previous sea-going experience.

The coding of the equations was more straightforward, once the previous steps were accomplished. The goal, again, was given an ordered heading, determine the rudder required to get to the new heading. Also compute the current heading, velocity vectors and position, used for rendering the models.

The first step was to solve for rudder angle. Next was delta heading, $\psi - \psi_{com}$, the difference between the current heading and the ordered heading. If the delta heading was less than 180 degrees, the direction of the rudder is left, greater than 180, the rudder was to the right. The rudder angle, $\delta$, is then determined using the aforementioned

equation. The new angular velocity is obtained by adding to the old inputted angular velocity, the value from the fundamental turning dynamics equation, multiplied by time change since last update.

The velocity vector is obtained using the standard method of the speed multiplied times the appropriate trigometric function of the heading. Euler methods were used to determine the position vector, Figure 5.

### 3. Summary

This method is an excellent way of providing realistic movement of surface ship models. The models display advance and transfer, not unrealistic instantaneous motion. If the variables for each specific ship type were known, the motion will be on par with a ship simulator. Future work in this arena should include research into calculating roll, accounting for speed loss during turns, and lateral dynamics. Future topics should also further the realism by allowing the OOD to order only rudder angles or combinations of rudder angles and ordered course.

## D. PROPULSION DYNAMICS

The arrangement for controlling the speed of the ship, is the same as for controlling the course. CLIPS performs the high level decision making and C carries out the orders to accelerate or decelerate to reach the ordered speed. The function, *compute_dynamic_speed*, is designed for this purpose.

### 1. Theory

The background investigation for propulsion dynamics was identical to that of turning dynamics. The theoretical equations were provided by [PNA88] with guidance from Professor Papoulias. The fundamental propulsion dynamics equation is $\dot{u} = au + bn$ where $\dot{u}$ is the acceleration, $a$ is the ship's acceleration responsiveness, $u$ is the actual speed, $b$ is the strength of the propulsion plant and $n$ is the propeller revolutions per minute (rpm). The propeller rpm, $n$, is determined by the equation $n = k(u - u_{com}) + k_0$ where $k$ is

```
void turn_to_ordered_heading(float *turning_rate, float *heading, float *in_x, float *in_z,
                        float *in_speed, float *rudder, float *turning_response, float *rudder_strength,
                        float *rudder_damping_var, float *ordered_heading, float *vel)
{
int i;                          /* counter variable */
float pos[3];                   /* position */
float rudder_coeff = 1.0;/* rudder coefficient, one deg of rudder per one deg of heading change*/
float ordered_head_radians;
float delta_head;               /* difference between ordered & actual */

/* convert degrees to radians */
*heading *= DEGREES_TO_RAD;
*turning_rate      *= DEGREES_TO_RAD;
ordered_head_radians = *ordered_heading * DEGREES_TO_RAD;

 /* rudder measurements are in radians */
delta_head = (*heading - ordered_head_radians);
if (delta_head < 0.0) delta_head += 360.0 * DEGREES_TO_RAD;
if  (delta_head <= 180.0 * DEGREES_TO_RAD)
                *rudder =  rudder_coeff * delta_head + (*rudder_damping_var * *turning_rate);
else
                *rudder = - rudder_coeff * delta_head + (*rudder_damping_var * *turning_rate);

/* full rudder = 30 degrees */
if (*rudder > 30.0  * DEGREES_TO_RAD) *rudder =  30.0 * DEGREES_TO_RAD;
if (*rudder < -30.0 * DEGREES_TO_RAD) *rudder = -30.0 * DEGREES_TO_RAD;

/* dynamic calculations for heading and angular velocity (turning rate) */
*turning_rate = *turning_rate + ((*turning_response * *turning_rate) +
                        (*rudder_strength * *rudder)) * ship_delta_time;
*heading = *heading + (*turning_rate * ship_delta_time);

/* Euler method for finding position using velocities */
/* no acceleration is considered */
vel[X] = *in_speed * KTS_TO_YARDS_SEC * fcos(*heading);
vel[Y] = *in_speed;
vel[Z] = *in_speed * KTS_TO_YARDS_SEC * fsin(*heading);

/* compute new positions */
for (i = X; i <= Z; i++)
   pos[i] = vel[i] * ship_delta_time;

/* pass out the new x, z values */
  *in_x += pos[X]; *in_z += pos[Z];
} /* turn_to_ordered_heading */
```

**Figure 5 - Dynamic Turning Function**

the feed back, corrective rudder coefficient used in turning dynamics, $u_{com}$ is the ordered or commanded speed and $k_0$ is the feed forward, predictive rudder coefficient. The feed forward coefficient, $k_0$, can be further calculated by the equation $k_0 = u_{com} \times \frac{n_{max}}{u_{max}}$ where $n_{max}$ is the maximum propeller rpm and $u_{max}$ is the maximum speed of the ship.

## 2. Implementation

The Officer of the Deck's orders to the lee helmsman have been simplified, similar to the helmsman orders. The OOD in this implementation only gives the desired speed, not other engine orders. The goal of the function *compute_dynamic_speed* is to receive an ordered speed and calculate a current speed. This function removes any instant acceleration and deceleration from the rendered models. The output is a smooth transition from one speed to the next.

The coding required for the propulsion dynamics implementation was mainly straightforward assignment statements. However, assumptions were again made to the value of the coefficients, $a$, $u_{max}$, and $n_{max}$. The maximum speed and rpms were obtained from previous shipboard experiences. However, the ship's responsiveness was the one variable based on the least educated guess. The function uses the Euler method to find velocities based on accelerations, Figure 6.

## 3. Summary

The dynamic propulsion function works fairly well. The models behave in a more controlled manner. The problem of instant deceleration or acceleration has been solved, which was a major goal. To render models suitable for use in a simulator, much more work must be done to obtain accurate numbers for the variables used in this function. The major problem was determining the ship's responsiveness variable. Unlike other variables where a formula could be used to obtain a relationship or others which could be extracted from shipboard experience, this variable was a best guess approximation. Future work could be

```
void compute_dynamic_speed(float *actual_speed, float *ordered_speed)
{
float     spd_max = 25.0;                          /* max spd */
float     rpm, rpm_max = 100.0;                    /* max rpm */
float     acceleration_responsiveness = -0.03,
          propulsion_plant_strength,
          feed_back_rudder_coef = -4.0,            /* assumes 4 rpms per 1 kt of spd */
          feed_fwd_rudder_coef,
          u_dot;                                   /* acceleration */

feed_fwd_rudder_coef = *ordered_speed * rpm_max / spd_max;

propulsion_plant_strength = - acceleration_responsiveness * spd_max / rpm_max ;

rpm = feed_back_rudder_coef * (*actual_speed - *ordered_speed) + feed_fwd_rudder_coef;

u_dot = (*actual_speed * acceleration_responsiveness) + (propulsion_plant_strength * rpm);

*actual_speed = u_dot * ship_delta_time + *actual_speed;
}
```

**Figure 6 - Propulsion Dynamics Function**

done which more accurately simulates the OOD's orders. This could include the desired engine direction and desired revolutions.

# VI. EXPERT SYSTEMS IMPLEMENTATION

There are literally hundreds of expert systems tools available commercially, ranging in price from under $1,000 up to $50,000. There are forward and backward chaining models available for almost any purpose imaginable. Examples of successful commercial expert system tools are for petroleum exploration, mineral ore deposits exploration, and financial planning. [Walk90]

In determining which inference engine to use in constructing an autonomous naval force, consideration had to be given to all the features discussed in the previous chapter. Are rules or frames best suited for representing knowledge? Have the ships attained a goal and want to know how they got there (the goal driven problem) or are the ships reacting to the environment (the data-driven problem). What type of communication link is available between the expert system shell and the graphics code? As discussed, CLIPS's control structure is forward chaining. Its knowledge base is rule based and was designed for maximum interoperability with the C programming language. In the fleet, ships contain many sensors, such as radars, sonars, and communications equipment, that continuously update the current situation. Sailors make decisions based on this sensor input. This is the classic data-driven problem. Therefore, CLIPS was the logical choice for my inference engine to model naval vessels at sea.

## A. CLIPS INTERFACE WITH THE GRAPHICS ENVIRONMENT

Determining which programming paradigm to be the controlling agent was an important consideration in constructing this autonomous world. There had been previous work using an imperative language to call CLIPS functions [Hopp92]. But this type of setup made the imperative language the controlling language. In order to fully utilize the capabilities of an expert system, it should be at the highest level, controlling the action. Therefore the expert system shell, CLIPS, was installed as the decision maker and thus able

to dictate the actions of the graphical models with C doing the actual computations and graphics rendering.

The work done for this thesis is the initial attempt at implementing an expert systems' driven graphical world. Therefore the steps taken to accomplish this task are delineated in detail to facilitate possible future work in this area.

The first step of implementation is the Makefile. There are two different approaches to linking CLIPS. One is to link to the original code installed, the other is to copy the code to your own directory and link it there. The former's advantages include not requiring additional memory for storage and immediate updates if a new version of CLIPS is installed. The latter was recommended in [Giar91] for ease of linking the two programs. Appendix A contains the Makefile used in this thesis work. This Makefile links to the original CLIPS source code installed on the IRIS.

Development of the communications link between CLIPS and the existing functional graphics program was a major accomplishment for establishing an expert system as the server and the graphical models as the clients. The requirements for information transfer between the two programming languages was critical. Neither server or client could function properly without timely data transfer.

The executable command, *gr_clips*, is created during the make operation. When *gr_clips* is typed at the command prompt, the CLIPS prompt is displayed. The CLIPS code has been modified to allow communications between the expert system shell and the graphics code. After the applicable rules are loaded at the CLIPS prompt, the autonomous surface ships are displayed.

The intermediatory required for connecting CLIPS and C is the main.c file. Its purpose is to inform CLIPS of any user defined functions. Figure 7 shows an example of the user defined functions in main.c. The various user defined functions are discussed in more detail below. They are the key to tailoring the communication's link between the client and the server.

```
/* user defined functions are described to CLIPS by the function UserFunctions() */
UserFunctions()
{
  /* external C function which computes the true bearing and returns a value of
  type double */
  extern double calculate_true_bearing();
  /* external C function which builds a multivalue field and passed the
  information to CLIPS */
  extern VOID cruiser_values();

  /* The first argument to DefineFunction is the name that CLIPS will use when
   invoking the funtion. The second argument is the type of parameter which is
   returned to CLIPS, (double -> d, multifield value -> m). The third argument is a
   pointer to the actual function. The fourth argument is a string representation of
   the third argument */
  DefineFunction("calculate_true_bearing",'d',calculate_true_bearing,"
  calculate_true_bearing");
  DefineFunction("cruiser_values",'m',cruiser_values,"cruiser_values");
}
```

**Figure 7 - Excerpt from main.c**

In order to modify the original C graphics program to work with CLIPS, two straightforward modifications were made. First, a new C function called *initialize_ship* was created. *Initialize_ship* contained all the code that occurred before the "while true" loop in the original graphics code. The other C function, *ship*, contained the remainder of the C code with the "while true" loop removed. The "while true" loop's purpose was to provide an infinite loop, which is now done in CLIPS.

The first CLIPS rule fired is *init-ship*, which calls the *initialize_ship* function. Its purpose is to initialize variables and graphical models. The second rule fired calls ship.c until the user exits the program. This is accomplished by designing an infinite loop in

CLIPS. At this point, the graphics program functions in the exact same manner with CLIPS interfacing with C as it did as a stand alone C program.

The user defined functions are the mechanisms used to transfer information between client and server. The client sends data to the server who processes the data through the rule based knowledge base and recommends actions dependent on the conditions met and rules fired. The recommendations made are in the form of modifying global variables pertaining to the ship's heading and speed. The expert system orders the new heading and speed, which is then processed inside the C program to determine the actual values assigned for each ship during that cycle. Meticulous attention must be given to ensure only one rule per cycle can be fired otherwise unexplained/unexpected results will occur.

## B. DEVELOPING AUTONOMOUS WORLDS

The most logical question now is in which direction should efforts be expended. Two different naval autonomous worlds were developed. In the development of each world, the author functioned as the knowledge engineer, because of my computer skills, and also as the expert, based on my training as a surface warfare officer (ship driver).

The first world created an autonomous naval force that simulated maintaining station in a defined maneuvering box. This is a common situation for ships at sea and a good beginning in building "smart" ships. Implemented in these "box" rules were two ships, an Aegis cruiser and an aircraft carrier, and one helicopter. All three vessels used the same type rules for maneuvering. Simple collision detection and avoidance was installed for the ships. The helicopter had the capability to land on the carrier and hover over the cruiser in addition to following the box rules. This simple world provided the opportunity to experiment with the different communications protocols available between CLIPS and C.

The next project was to develop an autonomous naval component to sail within a predefined lake. Three ships were incorporated into this world. Their goal was to not run aground and not collide with each other. The motivation behind this project was for its incorporation into NPSNET IV, thus providing the first naval autonomous agents in that

simulator. Three ships were rendered in the test platform as an Aegis cruiser, an aircraft carrier and a replenishment ship, AOR. After being networked to NPSNET, the three ships were rendered as sailboats.

The final rules developed were for a surface to air missile fired from the Aegis cruiser. The missile is not physically based, but is intelligent. The CLIPS rules will in most cases guide the missile onto the target.

The remainder of this chapter discusses in detail the user defined functions used for both worlds, the rules required for maintaining station in a box, the helicopter control rules and the rules used for the lake. The details are discussed to show the manner in which the expert system maintains control of the forces and it also shows the development from learning to correctly define user functions, to a simple autonomous world, and finally to the complex problem presented in the lake rules section.

## C. INTEGRATING CLIPS WITH EXTERNAL FUNCTIONS

As stated above, user defined functions are the mechanisms used to transfer data between CLIPS and C. Some examples of functions implemented for this work are functions to automatically send data to CLIPS every cycle. Functions called by CLIPS rules are used to provide the information required to perform the decision making process. Yet other external functions are called to modify ship's variables such as heading and speed. Both autonomous worlds essentially use the same user defined functions.

The key to successful implementation of the user defined functions is to understand how CLIPS passes arguments to these external functions. In C, arguments are listed directly following a function name within a function call. CLIPS actually calls the function without any arguments. Figure 8 gives an example of CLIPS invoking the function to determine the true bearing from the cruiser to the carrier. The arguments are stored internally by CLIPS and can be accessed by calling the argument access functions. Access functions are provided to determine both the number and types of arguments. [NASA91] The user

```
(bind ?find-true-bearing (mv-append ?cg-x ?cg-z ?cv-x ?cv-z))
(bind ?true-bearing (calculate_true_bearing ?find-true-bearing))

/* ?find-true-bearing contains the arguments passed */
/* ?true-bearing is the returned value from the external function
calculate_true_bearing which took the four inputs, cruiser's x, z coord
and carrier's x,z coord */
```

**Figure 8 - Example of CLIPS Invoking External Function**

defined functions are written to provide error checking, thus ensuring the correct type and proper number of arguments are passed.

CLIPS is designed to accept and pass symbols, strings, instance names, floats, integers and even unknown data types. There are two primary methods used in this work to pass values, single value and multifield value. Returning a single value is much the same as returning an integer or float in C. The multifield value can be thought of as an one dimensional array. It is a very powerful and useful method of passing arguments. The multifield value is simple to construct, requiring only assignment statements, such as those in C used to fill an array. The multifield value may contain any type of argument listed above and contain one or more arguments. The external function may then index the multifield value in a manner similar to indexing an array in C, and make assignment statements, perform calculations, etc. This method proved to be very valuable way of passing large amounts of data conveniently.

### 1. Returning Values to CLIPS from External Functions

Returning arguments from external functions to CLIPS was implemented as described above. The multifield value was used primarily to pass ship's status information at the beginning of each cycle. As part of the initialization process, the rules required the most up-to-date information about the vessels in the world. Therefore, at the beginning of

the loop, external functions were called that contained the required data. Some examples of these external functions are *cruiser_values, carrier_values, aor_values* and *helo_values*. The actual code may be found in Appendix D. These functions are called by CLIPS during each graphics cycle and give the most current positional, heading and speed information.

An example of an external function which returns a single value to CLIPS is *compute_true_bearing*. This function receives a multifield value, performs the calculations and returns a single value. The same calculation could have been performed in CLIPS, but because it is an interpretive language, CLIPS is designed as a decision maker. Thus the preferred method is to pass the arguments to an imperative language, which is designed to process deterministic mathematical functions, perform the calculations and return the result for the expert system's use.

### 2. Passing Arguments from CLIPS to External Functions

As was the case above, single and multifield values may be passed from CLIPS to external functions. However during this work, there was not an instance where the need for transferring one value was needed, it was always more. Therefore, it was necessary before every external function call to construct a multifield value and then call the necessary function. For example, to change the ship's heading and speed, the external function, *change_heading*, required three pieces of information. It needed the vessel's heading to be changed, the ordered heading and ordered speed. After receiving the orders from CLIPS, it modified the appropriate vessel's ordered heading and ordered speed. Other functions internal to the C program calculated the actual values for heading and speed until both were equal. Another similar example is the *move_helo* function. It receives orders from CLIPS for the helicopter and determines which values to assign to heading and speed based on whether it is landing on the carrier, hovering over the cruiser or flying independently.

The *change_heading* and *move_helo* functions receive orders from CLIPS and performed modifications to graphics global variables. This method and the example from

the previous section of *compute_true_bearing* are the two most common uses of passing information from CLIPS to an external function.

## D. BOX RULES

The *box* rules are the first attempt at developing an autonomous surface navy force. The motivation behind designing this system is based on the long standing naval tradition of maintaining *Gonzo Station*. Translated, given a grid or box, sail around inside but do not stray out of the assigned area except if necessary to avoid danger. The surface participants were an Aegis cruiser and an aircraft carrier. The high level goal for the expert system, was to process the ship's positional information, x and z coordinates, heading and speed, and react correctly to the situation. The expert system first determined whether a risk of collision at sea was imminent. If there is an impending collision, then orders were given to maneuver to avoid the other ship. If not, then determine if a maneuver was required to maintain station in the box otherwise maintain current course and speed. Figure 9 illustrates the CLIPS control sequence process.

### 1. User Defined Functions

Several C functions were developed to solve these problems. The function names correspond to calculations done aboard ships when solving maneuvering board problems to remain on station, tracking contacts, or maneuvering to avoid another ship. An excellent source of information concerning maneuvering board computations is [SWOS85]. The C functions are then imbedded into a user defined function, which is then invoked by CLIPS.

For a surface warfare officer, it is simple to find the true bearing to another ship. He can look through the compass rose and get the true line of bearing to the ship or get the information from the radar repeater. However since we have none of these capabilities built into the Silicon Graphics machines, a mathematical solution must be found. To find the true bearing between two ships, we must find the angle that the object of interest presents in relation to the x and z planes from our ship. The y plane is not significant because ship's

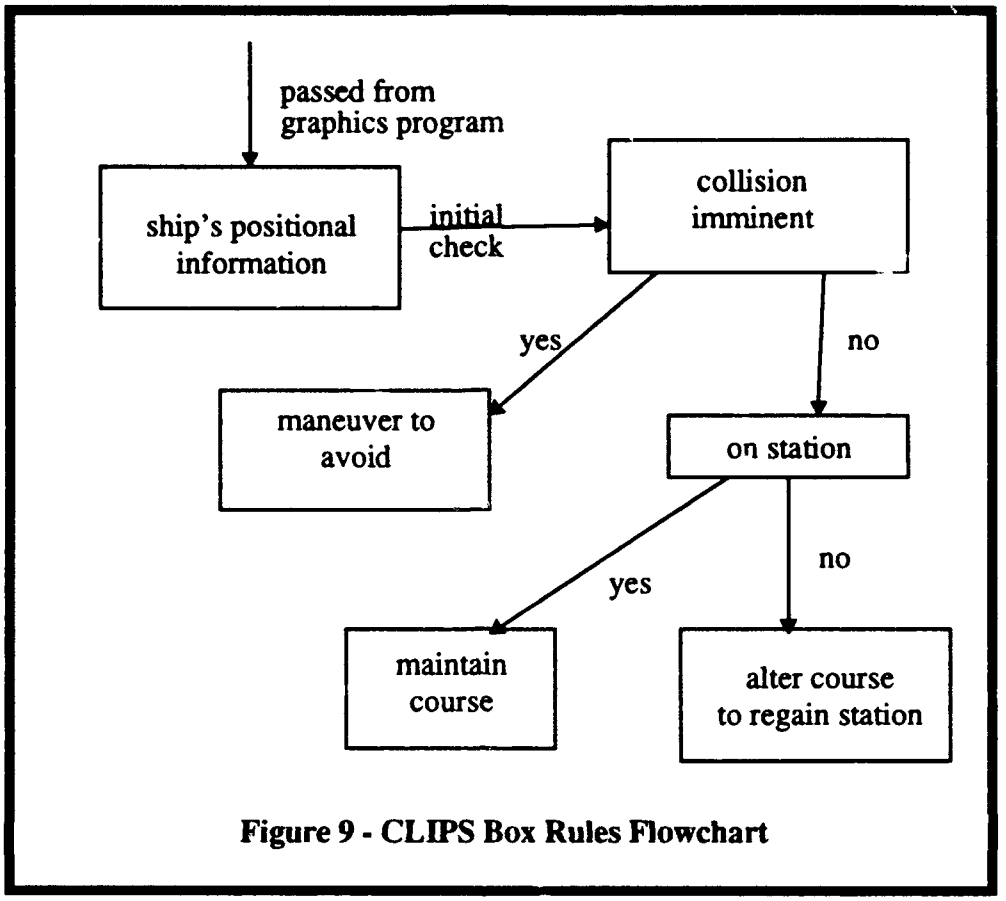



**Figure 9 - CLIPS Box Rules Flowchart**

altitude is constant, at sea level. The function *compute_true_bearing* calculates this angle with the following equation:

$$\text{true\_bearing} = \text{atan}\ \left(\frac{z2 - z1}{x2 - x1}\right) \times \frac{180}{\pi} \qquad \text{(Eq 1)}$$

where x1, z1 and x2, z2 are the x and z coordinates of our ship and the target ship, respectively. The relative bearing is calculated in the function *compute_relative_bearing* by the following:

relative bearing = true bearing - ship's heading. (Eq 2)

To calculate the range, the function *compute_range* uses the distance equation between points in a plane. . x1, z1 and x2, z2 again representing the coordinates of our ships, the distance formula is:

$$range = \sqrt{(x2 - x1)^2 + (z2 - z1)^2} \qquad \text{(Eq 3)}$$

These three functions were the building blocks for developing rules and functions to perform realistic simulations between the graphical models in the virtual environment.

## 2. Maneuvering Determination

During each cycle through the graphics loop, two important types of information are passed from C to CLIPS. One is the relative bearing and range between the two ships, the other is the current positional values. The rules to calculate whether a risk of collision exists are given highest priority and thus processed first. In this initial attempt at collision detection and avoidance, tests were conducted based on relative bearing and distance from each other, Figure 10. The reason for this methodology was that if a target ship is directly

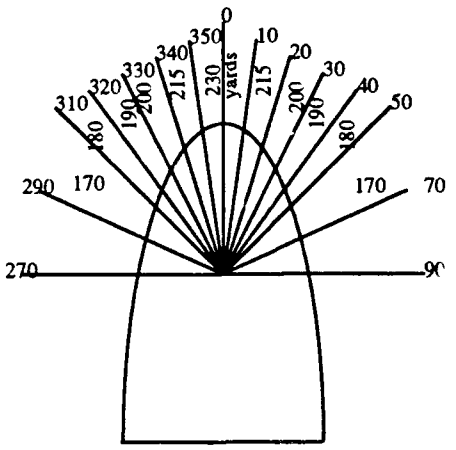



**Figure 10 - Relative Bearings and Ranges used for Collision Avoidance**

ahead of your ship, then there is a higher risk of collision and thus the flag is set even if the range is relatively large. As the target ship proceeds down either side, the range between

vessels becomes smaller before the collision flag is set. An example of this is if the target ship bears between 340 and 350 degrees relative, and the range is 230 yards the collision flag is set. However if the target ship is between 290 and 310 degrees relative, the range would have to be less than 170 yards to set the collision flag. This is a crude method of determining collision probability, however it was remarkably successful because of the speed of the program cycle, this was computed between 20 to 30 times a second. If any of these conditions were satisfied, then a boolean flag was set that allowed only collision avoidance maneuvering to occur, station keeping became a secondary issue. If the flag remained false and no risk of collision was present then CLIPS proceeded onward to determine whether the ships were on station or not.

If the collision flag is set, CLIPS orders the ship to turn away from the oncoming ship. If the target ship is on the port side, maneuver to starboard. If the contact is on the starboard side, turn to port. These rules do not conform to the international rules of the road [USCG83], but they do incorporate some collision avoidance capability. The maneuvering worked moderately well, however any collision between ships is unacceptable, therefore improvements needed to be made. In a later section, the improved collision detection and avoidance techniques are discussed, and they do conform with the rules of the road.

If the collision flag is not set, then a fact is asserted for each ship, containing the ship type and the positional information. There are eight rules corresponding to the eight maneuvering boxes shown in Figure 11. If the ship is inside of one of the maneuvering boxes, numbered one through eight, then the appropriate rule is fired and a new heading is ordered which will turn the ship to the left and orient it back towards the center of the maneuvering grid.

This process is repeated every cycle. Care must be taken to ensure that all unused facts are removed from the stack at the end of each cycle. This prevents calculations from using old data, thus displaying unexpected results.

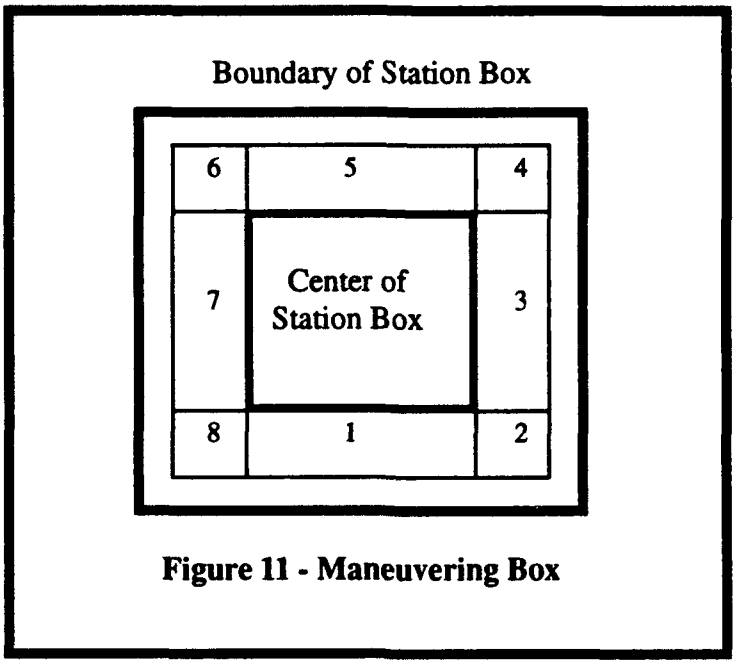



**Figure 11 - Maneuvering Box**

## E. HELICOPTER CONTROL RULES

In addition to the two ships, there existed a helicopter in the naval forces virtual world. This added more realism to the world, since most ships in the U.S. Navy are equipped to operate rotary winged aircraft. The features of the helicopter in this scenario are threefold. One the helicopter can transit in the same maneuvering box as the ships, or it can fly to the cruiser and hover over the its flight deck. Additionally, the helo can fly to, land and launch from the deck of the aircraft carrier. The setup is essentially the same as with controlling the ships using the box rules, CLIPS determines which course of action is required and orders C to carry out the assignment. Figure 12 shows the decision matrix used when deciding which set of rules the helicopter should follow.

Control of the helicopter is available to the user via a menu selection. The user can decide which vessel the helicopter should fly towards or if the helicopter should be launched from the vessel it is currently visiting. This flag is sent to CLIPS along with the aircraft's positional data every cycle utilizing the user defined function *helo_values*. Inside of CLIPS, the initial step is to bind the flag into a fact, which is then evaluated against the

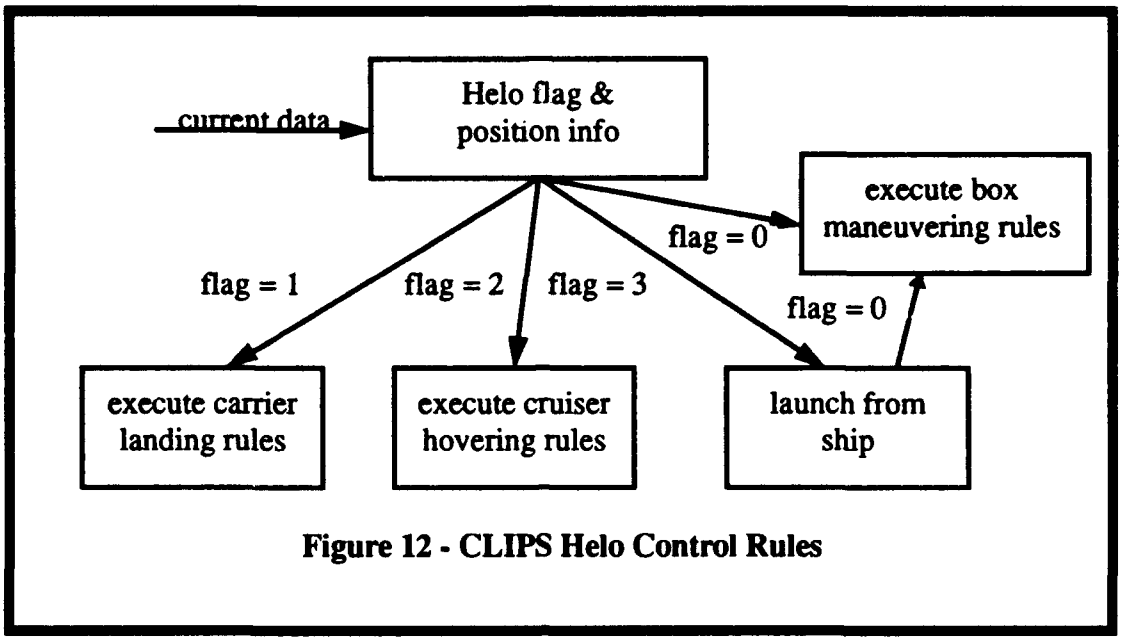



**Figure 12 - CLIPS Helo Control Rules**

rules. If flag is zero then no landing or launching operations are required. The helicopter flies according to the same set of maneuvering box rules as the ships abide by. The only difference from the ship's box rules is the size of the helicopter's maneuvering boxes are larger.

If the order is to land on the aircraft carrier, CLIPS performs two functions. One, it calculates whether the helicopter is currently over the flight deck of the carrier. This computation is done by comparing the ship's x and z coordinates against the x and z coordinates of the helicopter. This test determines if the two are within the same area in the x-z plane. The ship's y-position is constant, therefore the helicopter's y-position (altitude) is adjusted later when landing on the flight deck. If the helicopter is over the carrier's flight deck, then CLIPS orders the helicopter to match the course and speed of the carrier. Otherwise it calls the user defined external function, *calculate_true_bearing* to determine the direction to the carrier. CLIPS orders the helicopter to turn the shortest distance towards the carrier, increase speed to three times that of the carrier and fly towards its flight deck. This process is repeated until the user selects another option from the helicopter control menu.

These rules work quite well in controlling the helicopter. The external user function, *move_helo*, contains all the required code to "smooth" the helicopter's transition towards and the landing on the carrier's flight deck. When the *land on carrier* option is selected, the users sees a smooth turn towards the ship and a gradual increase in speed until the aircraft is over the flight deck and its heading is that of the carrier. If the approach leaves the helicopter over the flight deck, but not heading in the same direction as the carrier, the helicopter will fly away and make its approach from a more desirable direction. Once over deck and heading in the correct direction, the helicopter decreases altitude and speed to land on the fli it deck and match the carrier's speed. The helo will remain on deck by matching the course and speed of the carrier. The visual effects are quite impressive, especially when the carrier is turning.

To launch from the carrier, the user will select that option from the menu. A launch order is given and C passes that information to CLIPS, which processes the new information and fires the appropriate rule. The order is given to C, to increase altitude and speed to predetermined values. Once these values are attained the helicopter reverts back to flying the box rules and continues to do so until another helo control option is selected from the menu.

The user also has the option to hover over the flight deck of the cruiser. The functions used are exactly the same, except CLIPS will send orders and data relating to the cruiser instead of the carrier to the C user defined functions.

## F. LAKE RULES

After constructing an initial autonomous naval force, the next step was to add more realism to the virtual world. The motivating factor in developing the lake rules series was for its eventual incorporation into NPSNET IV, which was displayed at The Tomorrow's Reality Gallery at SIGGRAPH in August, 1993. The plan was to develop a stand-alone test platform, in order to develop, evaluate and refine the rules and then incorporate the naval forces under the autonomous vehicles control program in NPSNET.

The operating environment for the surface ships was a lake, designed by R.D. Young, with an island in the southwest corner. The goal was to have three ships sailing on random courses within the lake. The ships should avoid crossing outside the boundaries of the lake and avoid colliding with one another, Figure 13. The problem of a real-time solution for

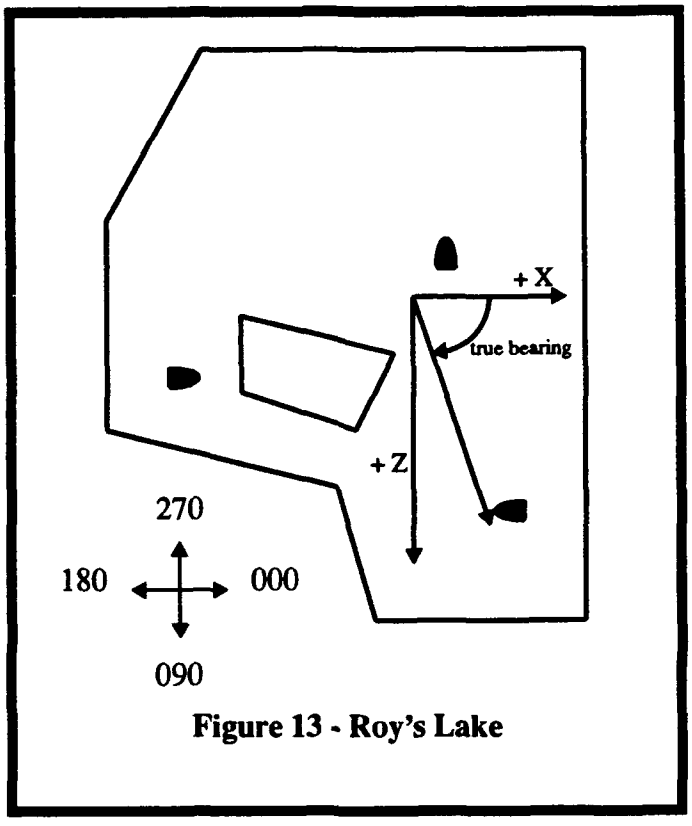



**Figure 13 - Roy's Lake**

detecting the boundaries of the lake was not a trivial problem. Also, a more realistic collision and avoidance system must be installed using the guidelines of [USCG83].

### 1. Collision Detection

In developing a mechanism for detecting the risk of collision with a boundary or another object in the world, the thoughts were to develop a system that would work for both static objects (lake boundaries) and moving objects (ships). We decided to tackle the problem by using geometry, namely various formulas for the equation of a line. The following example shows what the process is in determining whether a ship's present course will cause her to intersect with a lake boundary. The required inputs for these

calculations are the ship's x, z coordinates and heading and the x, z coordinates for the two endpoints of the boundary line. This method is the foundation for all collision detection and subsequent collision avoidance actions taken throughout this world.

Therefore, using [Swok79], to find the equation of a boundary line required that the endpoints be plugged into the point-slope form for the equation of a line formula, $z2 - z1 = m(x2 - x1)$, where m is the slope of the line. The z-intercept is then computed using the slope-intercept form for the equation of a line, $z = mx + b$, where b is the z-intercept. Now we have the equation of not only that line segment, but the equation of the infinite line of which the boundary segment is only a small part.

The other line segment was formed by the ship. One endpoint was the current x, z coordinates. The second endpoint was an arbitrary dead reckoned position, obtained by using the equations:

$$\text{ship\_dr\_x} = xposit + 1000 \times \cos(heading) \quad \text{and} \qquad \text{(Eq 4)}$$

$$\text{ship\_dr\_z} = zposit - 1000 \times \sin(heading) \qquad \text{(Eq 5)}$$

The equation of the line formed by the ship's position and its dead reckoned position was computed in the same manner as the boundary line segment.

Thus far we have obtained the slope and z-intercepts for the two line segments. The next step is determine if there is an intersection between the two lines. If an intersection exists anywhere on the infinite lines formed by each line segment, then the slope-intercept equations of the two lines must be equal. We then set these two equations equal to one another and solve for the x coordinate of the intersection, intercept_x, using Equation 6. Upon finding this value, the z coordinate of the intersection, intercept_z, is obtained by using the slope-intercept equation of Equation 7.

$$\text{intercept\_x} = \frac{(zintercept2 - zintercept1)}{(slope1 - slope2)} \quad \text{and} \qquad \text{(Eq 6)}$$

$$\text{intercept\_z} = \text{slope2} * \text{intercept\_x} + \text{zintercept2}. \qquad \text{(Eq 7)}$$

Now the intersection coordinates are known, a check must be made to determine whether the intercept point lies on either of the two lines. The intersection point, when calculated in this manner, will yield a value which may or may not be within the limits of the original coordinates. In other words, it may fall behind or ahead of the original line segment while still on the infinite line formed by the slope of the line segment. Therefore, must determine whether the intersection lies on both line segments. The method used when dealing with static lake boundaries follows. First determine which endpoint of the boundary line segment is the smallest x coordinate value, called minimum-x, with the other endpoint being the maximum-x. If the intercept_x falls within the range of minimum-x and maximum-x then the intersection point is on the two line segments and a valid intercept point exists.

The intercept computations described are contained in the external function *calculate_intercept*. The purpose of this function is determine the intersection and then return a distance to that intercept point. If a valid intercept point exists, it calls the *compute_range* function with the ship's coordinates and the intercept point's coordinates and returns the distance to that point. If a non-valid intercept point exists, then a large dummy value is assigned to the distance. This dummy value is larger than the diameter of the lake, therefore it will not be mistaken for a valid range.

A similar procedure was followed when calculating the closest point of approach (CPA) between two ships. The functions to determine risk of collision between ships and the recommended action was conducted in C. The information is then passed to CLIPS, therefore allowing it to decide if a risk of collision exists. CLIPS will then invoke the avoiding rules or invoke the normal transit rules as necessary. This arrangement allowed for quick computations of the CPA between each ship before interfacing with CLIPS.

The C function, *calculate_cpa* uses the same principles as the *calculate_intercept* function, with one significant difference, the method used to determine the validity of an intercept point. The inputs for this function are the two ship's heading and x, z coordinates. The line segments are both derived from the ship's information. The intercept calculations

are performed in the manner as in *calculate_intercept*. However, the check for a valid intersection point is different. The static boundaries method was tried, but the results were not always correct. Therefore, a different method of determining intersection point's validity was devised. After the intercept point was calculated, a relative bearing was computed to the point from both ships. If the point was forward of the beams on both ships, then the intercept point was valid. If each ship continued on its present course then a collision would occur. The range is computed for this point and passed back to the calling procedure. If the point is invalid, then a dummy value is assigned, again larger than the diameter of the lake.

This method of collision detection has provided excellent, predictable results in all cases. The design for the static with moving objects case and the case with two moving objects is essentially the same using simple geometry as described.

### 2. Collision Avoidance

After determining tat the possibility of collision exists and where the relative bearing and range to that intersection point lies, each ship must be equipped with the ability to properly maneuver to decrease the likelihood of collision rather than increasing that probability. Therefore, the nautical rules of the road were implemented. In the first draft of collision avoidance with the box rules, the ships simply turned away from a contact. This was not in accordance with [USCG83]. There are three general maneuvering cases addressed in this work:

- meeting,
- crossing, and
- overtaking.

These are the most common situations that occur at sea. A meeting situation is where two ships are on reciprocal or nearly reciprocal courses and the relative bearings to each other is off the bow. In a crossing situation, the contact lies forward of your beams and will cross your bow. In an overtaking situation, the ship is coming up from the stern of the contact. Figure 14 illustrates these concepts.

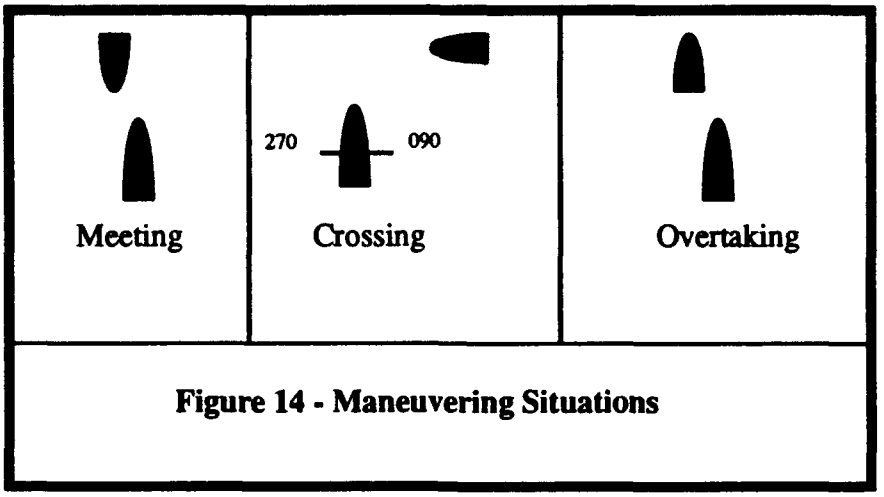



**Figure 14 - Maneuvering Situations**

In either case, there is a "stand-on" or privileged vessel and a "giveway" or burdened vessel. Each ship must determine what its current situation is and act accordingly. In a this computer simulation, each vessel knows exactly whether to stand-on or giveway based on how the program is written. In reality, ships do not always act as they should and the stand-on vessel may actual have to maneuver to avoid danger since the burdened vessel did not act properly. This feature is not included since the computer models recognize their situation and act correctly.

The function designed to make these determinations is *collision_avoidance*. The arguments to this function are the true bearing, relative bearing, target angle and range from our ship to the contact. Additionally the heading of each ship is passed. The target angle of the contact is our relative bearing from her. The function conducts tests on the inputted data and determines whether one of the three maneuvering cases exists. The range is considered to ensure maneuvering occurs before the ships are too close and also ensures that maneuvering is not done when the ships are far apart. *Collision_avoidance* returns the new heading. If none of the situations are satisfied, then the old heading is returned, and the ship continues to sail in the same direction.

The corrective action has been simplified and deals only with heading changes, not any adjustments in speed. In a meeting situation, both ships are burdened and should

alter course to starboard to allow safe passage between the two ships. The ordered heading is the current heading plus 30 degrees.

In a crossing situation, if the contact is on your port side, then you have the right of way and are the stand-on vessel. No maneuvering is required for the stand-on vessel. If the ship is on your starboard side, then you are burdened and must give way. The programmed response is to add 30 degrees to the contact's true bearing and go past her stern.

In an overtaking situation, the vessel being overtaken has the right of way. The burden vessel in this case, will add 30 degrees to its heading and overtake the contact on her starboard side.

These rules work quite well for two ships avoiding one another. However there are three ships in the world. Therefore a priority system is established. Using the cruiser as an example, it will first determine which closest point of approach (CPA) is closest, the CPA to the carrier or the AOR. If the carrier CPA is the smallest and less than a predetermined value, then a call is made to the *collision_avoidance* function to determine which course of action is required. If the AOR's CPA is the smallest, then *collision_avoidance* is invoked to determine if action is required. Therefore, the priority is placed on the contact which we might hit first. When all three vessels are close to each other, some interesting results have occurred, however only during a small percentage of the time have the ships hit each other.

### 3. A Loop around the Lake

Now that the functions are constructed for collision detection and collision avoidance, we discuss the CLIPS rules and the sequence of events leading to the ships transit around the lake. At the beginning of every cycle, a multivalue field is passed to CLIPS containing the collision status. If a possibility of collision exists, a new ordered heading will be included in the multivalue field. Maneuvering to avoid collision then takes the highest precedence. CLIPS orders the ship to turn to a new heading taking it out of

danger. If no risk of collision exists, then a new fact is asserted with the ship's positional information and CLIPS starts checking to determine if there exists a danger of coming too close to the lake boundary. This checklist is outlined in Figure 15.

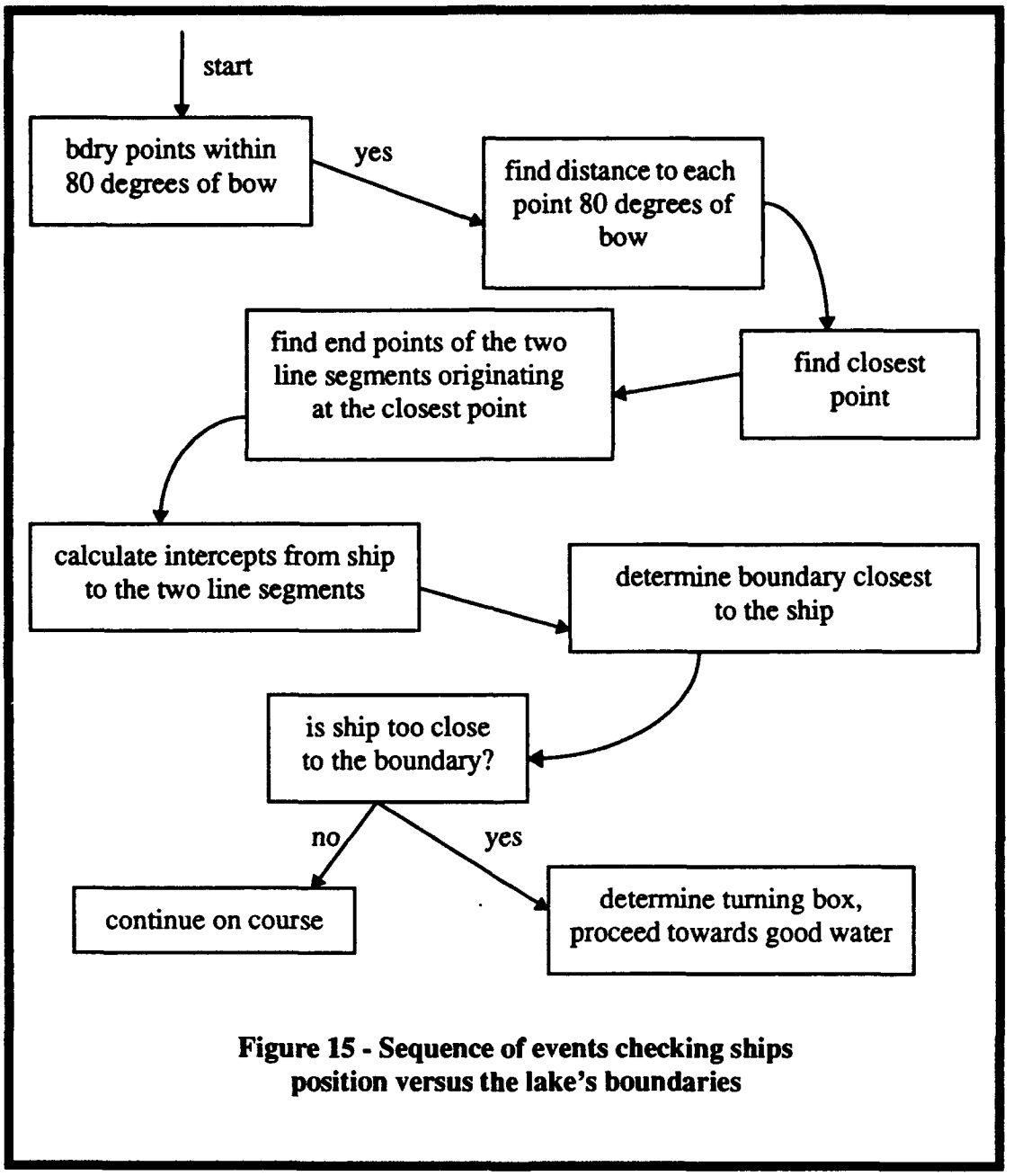



**Figure 15 - Sequence of events checking ships position versus the lake's boundaries**

There has to be a priority given between running aground and collision with another ship. Since collisions at sea are potentially more deadly and devastating, it was

given the highest priority. Therefore the first item of business that CLIPS attends to is checking for the collision flag. If the collision flag is set, then CLIPS orders the affected ship to maneuver to avoid collision. There are no boundary condition checks performed when the collision flag is set. The ability to look at the lake boundaries and evaluate the collision situation, would require the expert system to have "look ahead" capabilities, which are an order of magnitude harder than the reactive behavior problem addressed in this work. There have been a large number of test cases performed on this knowledge base, and rarely have we seen a ship run aground while trying to avoid another ship. This is due in part to the value assigned to "get no closer than" to the shore variable. Small values put the ships too close to shore and excessively large values never allowed ships to get close to shore. Therefore, an intermediate value must be determined through test case. The most common problem encountered, which the "look ahead" capability could conceivable solve, is the situation where two ships are on a parallel course, close together and side by side, going approximately the same speed and they reach a lake boundary at approximately the same time. One of the ships inevitably turns sooner and towards the other ship, creating a collision avoidance situation. The programmed response in the cases we have witnessed is for the burdened ship to turn towards the stern of the privileged vessel and pass behind. However in the cases where the initial distance was very close, less than a ship length, a collision has occurred while turning to the new ordered heading. This has not happened frequently, however, and ensuring the ship's speeds were different, avoided this problem almost entirely. This problem is unrealistic however, because ships do not steam alongside each other when close to shore.

If the collision flag is not set, then CLIPS has a large number of conditions to check. This is accomplished by asserting facts and firing rules, in sequence, and eliminating facts until a single conclusion is reached at the end of the process, Figure 15. The first step is for the ship to determine where the boundary points are in relation to the ship. This is done by invoking the *relative_bearing* function for all boundary points. The next rule eliminates the facts where the points are greater than 080 degrees relative and less than 280

degrees relative. The ship is now only concerned with points that are forward of its beams. The next rule computes the range to all points forward of the beam, by invoking the *calculate_distance* function. The point which is forward of the beam and closest to the ship is dubbed "closest point" and intercept calculations are performed using this point. The closest point is an endpoint for two line segments, therefore the other endpoints are needed. CLIPS evaluates the list of endpoints and determines the appropriate pair and invokes the *calculate_intercept* function to determine the distance to each of the line segments. After receiving the outputs and comparing the distances, CLIPS then keeps the smallest distance. This new smallest distance is the range to the lake boundary that the ship will come in contact with if no maneuvering is done. If this smallest distance is less than a predetermined "get no closer than" distance, then CLIPS orders the ship to turn. If the distance is greater, then no new course is ordered.

This process is repeated for every ship, every time through the loop. After refining these static object detection rules, we have not witnessed one instance of the ship running aground without provocation from another ship. Even though there are numerous calculations, the program still able to render the graphical models for three ships in real-time, even on an Indigo.

### 4. Where To Turn Now?

If CLIPS orders the ships to maneuver because they are within a predetermined range of the boundary, the ship determines its location in the lake relative to the island and either turns toward open water or traverses the small channel between the island and the southwest corner of the lake. To make this determination, a series of lake maneuvering boxes are constructed and the ship will follow the guidelines set forth by the rules for these maneuvering boxes, Figure 16.

The maneuvering boxes are simply overlays which assist in determining which direction for the ship to turn. They can be thought of as fences around a pasture. If at the fence, determine the direction to the center of the pasture and turn in that general direction.

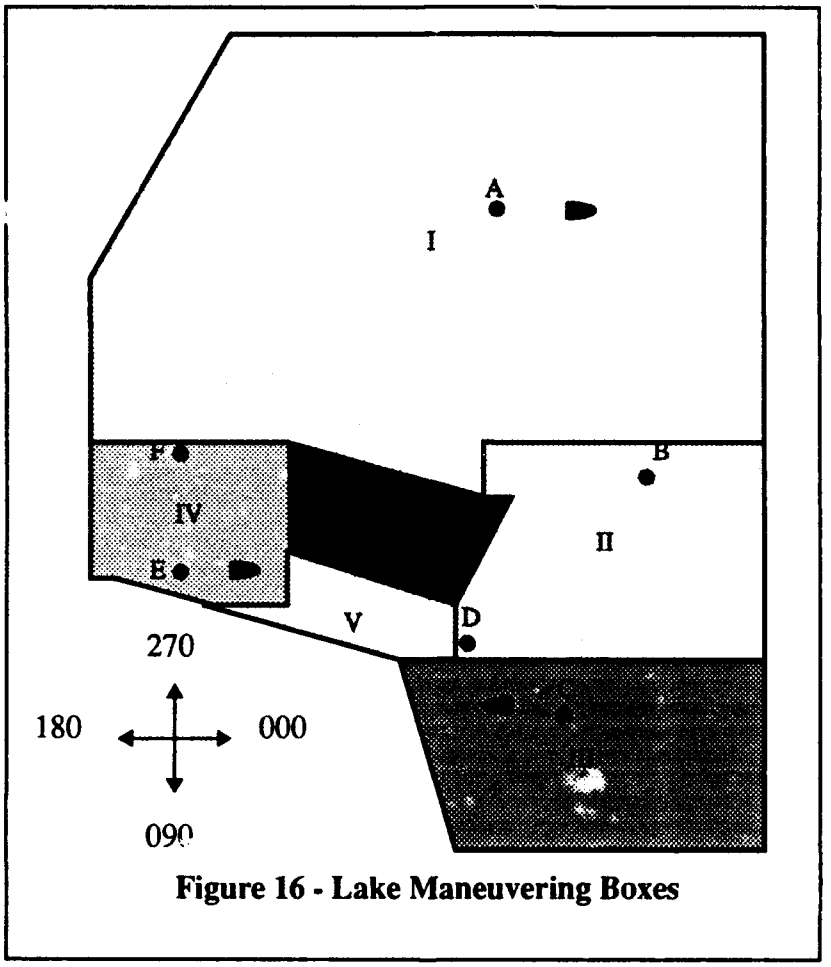



**Figure 16 - Lake Maneuvering Boxes**

*a. Open Water Maneuvering*

In maneuvering around Roy's Lake, five separate sets of fences have been constructed. When the ship is in the first box, I in Figure 16, and reaches a boundary of either the island or boundary of the lake, a call is made to the external function, *compute_relative_bearing*. The arguments passed are the ship's position and the coordinates of a point in the approximate center of maneuvering box one. After the bearing is obtained, an order is given to turn the ship. The direction to turn is determined by value of the relative bearing. If the value is between zero and 180 degrees, the point is on the starboard side and the ship turns right. If the value of the relative bearing is greater than 180 degrees, then the ship turns left. To add some randomness to the ship's movement, the new course is determined by dividing the relative bearing by three (or any small number) and

adding it to the ship's current head, to obtain the new ordered head. This method prevented the ships from always turning towards the center of the maneuvering box, towards open water. This provided a more interesting graphical display and tested the rules more ruggedly. Maneuvering boxes two and three work in the same manner as box one.

***b. Restricted Water Maneuvering***

Maneuvers around the island were slightly more complicated than open water maneuvering. When in restricted waters, most sailors do not want random courses for transiting through a channel. Therefore, the randomness features of rules one, two and three are removed. When a ship is in maneuvering box IV and encounters a boundary, a call is again made to *compute_relative_bearing*. However, the predetermined maneuvering point is not the center of the box. It is a point used as a beacon to guide the ship through the channel. If the ship is heading is approximately 340 to 160, the beacon used is beacon D, which is close to the boundary of maneuvering box V. CLIPS gives the order for the ship to proceed directly towards that point, with no randomness factors in the course determination. Conversely, if the ship is headed between 160 and 340, the maneuvering point or beacon is more towards the upper center of box IV, beacon F. This method of channel traversal allows the ships safe passage in restricted waters. This simulates many of the actions taken by actual ships which use such navigational aides as the buoy system, ranges, lighthouses, radio towers, stacks, church steeples and numerous other visual aides which are used to assist in determining a safe passage through restricted waters. Maneuvering through box V is conducted in the same manner as box IV.

## G. MISSILE CONTROL RULES

The last set of rules developed for this thesis work was for the implementation of an intelligent surface to air missile. The missile was designed to be launched from the Aegis cruiser's vertical launch system. The rational is to incorporate a self defense/strike capability into the virtual world and the Aegis cruiser is the logical choice for this assignment because that is its primary mission speciality in the Navy. The rendered

missile's flight is a smooth, noncartoonish motion, however it is not physically based. The heading and pitch of the missile are controlled by the expert system. CLIPS provided the required information for the missile to conduct a "tail chase" flight.

The expert system receives positional information on the potential target. The target, in this scenario, may be an enemy plane or missile. The order to fire is initiated by the user, by depressing the F(ire) key on the keyboard. The expert system calculates the true bearing and elevation of the target from the cruiser and orders the missile to engage. Upon launch, the missile flies towards the target's current location. This is the classic homing, tail chase pattern. The missile uses its superior speed to eventually overtake the target and impact it. Upon impact, the rules are halted and the missile is ready to fire again.

This is the only attempt at modeling and controlling a true three dimensional model in this thesis work. Even though the helicopter moves in three dimensions, it uses the same two dimensional rules the surface ships used. The altitude and pitch were handled as a special case. For this missile, Euler angles are used for computations on its positional data. Euler angles ignore the interaction of rotations about the separate x, y and z axes, which makes physically based missiles difficult using these angles [Watt92]. There is ongoing research at NPS on three dimensional, physically based spacecraft using quaternions [Hayn93]. Using quaternions eliminates the problems of Gimbal lock and orientation interpolation caused by Euler angles. The purpose of the missile rules and the helicopter rules, is not to dynamically model spacecraft, but to develop and introduce higher level rules to control them.

### 1. Initialization

At the beginning of every graphics cycle, the missile flag is check to determine whether a order has been placed to launch a missile. If no missile has been fired, then there is nothing to do and no missile rules are executed. If the missile has been launched, then a check is made to determine if the missile has impacted the target. The missile flag is true throughout the entire missile flight and this check determines if the missile is at the end of

its mission. If no impact yet, CLIPS receives target positional data from C. The expert system then calculates the bearing and elevation to the target by invoking the *calculate_true_bearing* external function. To find the target's true bearing, *calculate_true_bearing* is called with the x and z values, because bearing is the angle to the target in the x-z plane. The target's pitch (or elevation) in relation to ship is calculated by passing the x and y values. The pitch is the true bearing to the target in the x-y plane. Next CLIPS compares the returned values and determines the course of action for the missile to take.

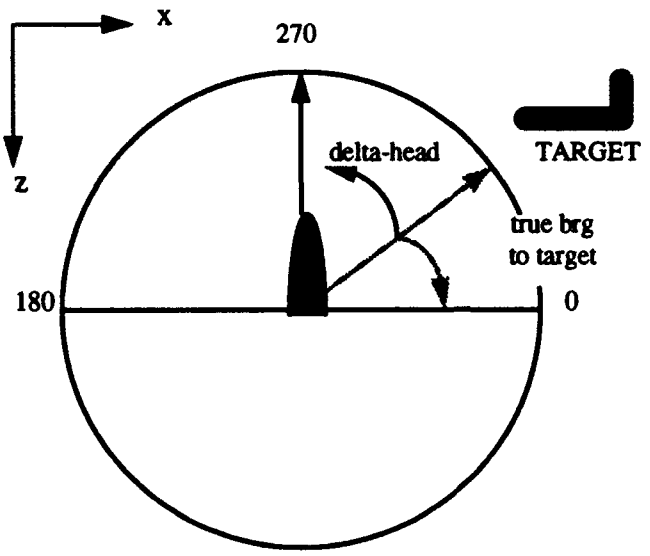
### 2. Missile Heading Rules

The missile heading rules have one purpose, turn the missile in the shortest path towards the target. This is accomplished in three steps. First the difference in the heading is calculated by the missile's current heading from the bearing to the target. The difference is called delta-head. This value is adjusted to ensure 360 modularity, if necessary. Based on the value of delta-head, a flag to turn either right or left is set. If delta-head is less than 180, the missile will turn right, if greater than 180 then the shortest path to the target is the left. Figure 17 shows an example of determining the missile's path using the shortest path algorithm.

### 3. Missile Pitch Rules

The missile pitch rules ensure the missile is pointed at the target. The missile is fired from a vertical launch system, this means that the missile must first travel straight up and then turn towards its target and begin the engagement. In this model, the missile rises initially 40 yards, before beginning its maneuver.

The missile's initial pitch is 90 degrees as it sits in the launcher and it remains constant after launch until its altitude is 120 feet. After reaching this height, the computations are made to determine which direction the missile should point. The manner in which this is conducted is as follows. The missile's pitch is compared to the angle of




1. find true bearing to target
2. delta head = bearing to target - missile's heading
3. ensure delta-head is modulo 360
4. if delta-head < 180 then turn-right
else turn-left

EXAMPLE:
1. true-bearing = 320
2. delta head = 320 - 270 = 50
3. ensure delta-head is modulo 360
4. shortest path from missile to target is right

**Figure 17 - Shortest Path Algorithm**

elevation to the target. The goal is to make them the same. A flag is set that either increments or decrements the pitch value.

### 4. Missile Flight Rules

The missile flight rules take the inputted flags for heading and pitch adjustments and orders the graphics program to orient the missile accordingly. There are four combinations of heading and pitch adjustments handled.

•turn right, decrement pitch

•turn right, increment pitch
•turn left, decrement pitch
•turn left, increment pitch

The external function *modify_missile_position* is used to change the ordered values of the missile's heading and pitch.

### 5. Summary

The intelligent missile incorporated into this virtual world provides the user with another feature common to naval surface ships which adds to the realism. No particular missile's characteristics are included in the design of the missile. However the rules are develop which allow the under appreciated surface warrior the satisfaction of hitting the target he intended to shoot.

## H. SUMMARY

This implementation of a rule based expert system controlling graphical models proves that autonomous agents are be controlled in real-time. The rules implemented for the surface ships is an actual representation of real world rules, thus the actions of the ships closely resemble real life. The foundation is in place for constructing worlds with autonomous agents of all types. Many of the external functions can be used for calculations not only for surface vessels, but for land-based vehicles and aircraft.

The surface has only been scratched when considering how this method of implementation can be used. The possibilities are endless for autonomous agents in a virtual world.

# VII. INCORPORATION INTO NPSNET IV

The original NPSNET was implemented as a land based battlefield simulator with no naval elements included. The latest version is NPSNET IV, which is object oriented, and contains autonomous agents and much more realistic graphics. Inclusion of this work of naval surface agents, provided NPSNET with a more realistic, joint approach to the battlespace simulator. The rules and external functions developed for the naval surface agents will enable future work to be done by using these as a foundation for any other more complex autonomous agents.

## A. DEVELOPING TEST PLATFORM

The test bed was developed in coordination with the designers of NPSNET IV. Their goal was to put autonomous boats on a lake. The test platform lake was built, in SGI coordinates, to the same proportions as the original lake. Using NPSOFF models, rules and functions were developed, as described in previous sections, to sail the boats around the lake without running aground or colliding with one another.

This arrangement allowed the autonomous agent developer the opportunity to work independently in devising and debugging his own virtual world. Only when convinced that each situation was handled correctly, was it prudent to connect the two systems.

## B. INTEGRATION

One of the major goals of NPSNET IV was to incorporate autonomous players into the environment. Therefore during the design phase, an interface mechanism was developed which enabled various autonomous agents the ability to interact in the environment. Agents could be selected via the menu. This action asserted facts which activated CLIPS rules and consequently activated the agents. The inclusion of this work on surface agents into NPSNET IV was the initial attempt at incorporating rules developed on one platform and merging them into NPSNET.

The NPSNET IV autonomous agent harness was designed in a similar manner to our test platform, in that the CLIPS rules were the high level decision maker calling external functions to perform calculations. Initially, the creator of the control harness had simple autonomous agents, with their corresponding rules, which proved that the network interface between NPSNET and the autonomous agents worked properly. Their hopes were to have autonomous agents developed independently and incorporated into their world.

To merge the code from our test platform into the autonomous agent control program, required several steps. First, all code which rendered the graphical models was removed[1]. Second, the CLIPS rules which provided for the infinite graphics loop are removed. This capability is provided for by the autonomous agents program.

Third, a function was developed with built in network compliance, which updated the master vehicles array in the autonomous agent controlling program. This function, *update_pf_boats*, receives a pointer to the vehicles array and performs two steps. It determines whether a Protocol Data Unit (PDU) needs to be sent over the network to NPSNET. For boats, an update is required if the boat has changed course or if more than five seconds has elapsed since the last update. For the missile, an update is sent every cycle because this is a short lived event, with continuous changes in heading and pitch. Next, *update_pf_boats* converts from the Silicon Graphics coordinate system to the Performer coordinate system and updates the vehicles array with updated boats and missile information. The final step is to merge our code and rules into the autonomous controlling program code. Figure 18 shows the differences between the SGI and Performer coordinate systems. Figure 19, is the code for *update_pf_boats*.

## C. PROBLEMS AND RECOMMENDATIONS

There were some stumbling blocks that should be avoided in the future. One is that the autonomous agent developer should use the same language as is used in NPSNET. The test platform was developed using K & R standard C. This was not done intentionally. Upon

1. NPSNET IV receives all vehicle data over the DIS network and renders the appropriate scene.

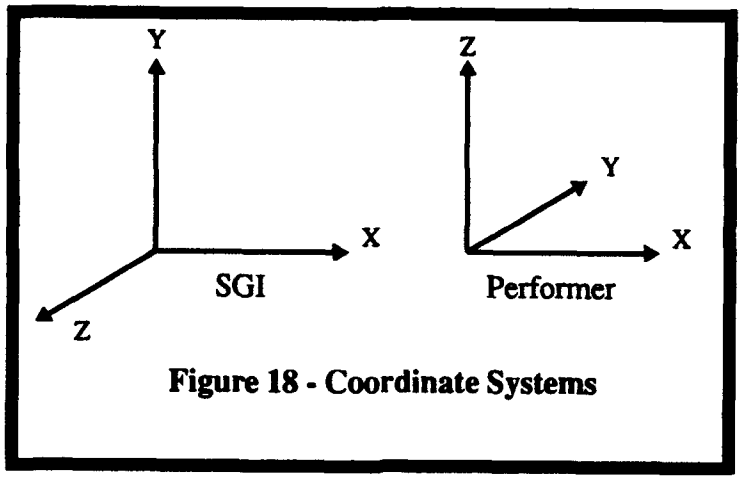



**Figure 18 - Coordinate Systems**

inspection, it was noticed that the Makefile contained the flag which activated this option. If the test platform is developed in C, then Ansi standard C should be used as a minimum. However, NPSNET uses C++, and if possible the test platform should do so also. With the autonomous vehicles controlling program in place and functioning well, any future test platform developers would be wise to develop their own platform using the same setup. This would make the transition from test bed to NPSNET much smoother.

The second stumbling block was the coordinate transformation. The optimal situation for the autonomous agents developer, would be to use the Performer coordinate system vice the SGI coordinates. This may be best accomplished by future graphics students learning on Performer instead of with the SGI coordinates. If this is not feasible, the transformations have been determined and are in *update_pf_boats*.

The idea of autonomous agents developers working independently of NPSNET is an excellent approach. This allows the developer to implement and debug the CLIPS rules and external functions, much more quickly than if they had to make the entire NPSNET code each time. The transition features are now in place to incorporate into NPSNET IV more

easily. The external functions are developed which should allow future developers to greatly improve the situations which the autonomous agents can handle.

```
/* send PDU update if heading changes, if greater than five seconds or if entity is a missile */
int update_pf_boats(int boat_num, VEHDATA* vptr)
{
int  send_update_flag = FALSE; /* time to send pdu update? */
static float boat_1_pdu  = 0.0,  boat_2_pdu  = 0.0, boat_3_pdu  = 0.0;
/* is Performer's heading different than our heading, if yes then update the network */
if ( (270 - (int)(vptr->posture.hpr[HEADING]) )!= (int)vessel[boat_num].heading )
                send_update_flag = TRUE;
  else /* if more than 5 seconds since last update, send update */
switch(boat_num)
{               case 0:  /* aegis */
                 if ( current_time - boat_1_pdu > 5.0) {
                   send_update_flag = TRUE;
                   boat_1_pdu = current_time;}
                 break;
                case 1:  /* cv */
                 if ( current_time - boat_2_pdu > 5.0) {
                   send_update_flag = TRUE;
                   boat_2_pdu = current_time;}
                 break;
                case 2:  /* aor */
                 if ( current_time - boat_3_pdu > 5.0) {
                   send_update_flag = TRUE;
                   boat_3_pdu = current_time;}
                 break;
                case 8:  /* missile */
                   send_update_flag = TRUE;
                 break;
}
/* update performer's vehicle array, Performer -> SGI */
vptr->posture.xyz[X] = vessel[boat_num].x_coord;
vptr->posture.xyz[Y] = - vessel[boat_num].z_coord;
vptr->posture.xyz[Z] = vessel[boat_num].y_coord;
vptr->posture.hpr[HEADING] =  270.0f - vessel[boat_num].heading;
vptr->posture.hpr[PITCH] = vessel[boat_num].pitch;
vptr->posture.hpr[ROLL] = vessel[boat_num].roll;
vptr->speed   = vessel[boat_num].speed;
vptr->vel3[X] = vessel[boat_num].vel[X];
vptr->vel3[Y] = - vessel[boat_num].vel[Z];
vptr->vel3[Z] = vessel[boat_num].vel[Y];

 return send_update_flag;
}
```

**Figure 19 - Update_Pf_Boats**

# VIII. CONCLUSIONS AND FUTURE WORK

## A. CONCLUSIONS

The primary purpose of this work was to prove the concept of controlling naval surface ships at-sea with an expert system and incorporating the results into NPSNET IV. Sub-areas of study included modeling the turning and propulsion dynamics of the surface ships.

After performing the development, testing and evaluation of the various features of this project, we have reached the following conclusions:

- A realistic, physically based simulation for surface ships is feasible.
- A rule based expert system is an excellent method for implementing autonomous agents into a virtual world.
- Incorporation into NPSNET IV is feasible.
- All of the above features can be done real-time.

## B. FUTURE WORK

Since this work is the first to use naval surface ships and an expert system with NPSNET IV, there is an unlimited number of areas where it can be expanded with future studies. The foundation is in place to develop CLIPS rules to handle other more complex situations ships may encounter at sea. Some of these include:

- Formation steaming with other autonomous ships. Various formations could include formations used for transiting, Anti-Submarine prosecution, and Search and Rescue.
- Underway replenishments rules to simulate refueling at-sea.
- Interaction with autonomous airplanes to include flight operations with fixed wing aircraft off the aircraft carrier.
- Physically based modeling of missiles.
- Future work in orders the expert system is capable of providing can be expanded. Example is ordering heading and speed changes when maneuvering to avoid other ships vice only heading changes.
- Future work in ship dynamics includes research into calculating roll, accounting for speed loss during turns, and lateral dynamics. Future topics could also further the realism by allowing the OOD to order only rudder angles or combinations of rudder angles and ordered course.
- Future work in propulsion dynamics includes more accurate simulation of the OOD's orders. This could include the desired engine

direction and desired revolutions.

This work is the first step in incorporating naval surface ships into NPSNET IV. It is also the first autonomous agents program developed independently and incorporated into the master controlling program for autonomous agents and networked into NPSNET. This thesis work proves the concept is valid, thus the possibilities for further enhancements both for naval elements and autonomous agents in general, are endless.

# APPENDIX A MAKEFILE FOR CLIPS DRIVEN GRAPHICS

```
#makefile to drive ship c code with clips call

GR_FLAGS = -O -I/n/elsie/work/zyda/rdobj3/lib -I.\
-I/n/elsie/work/zyda/imagesupport -cckr

# libraries needed for clips to link in unix
MORE_LIBS = -lm -lgl_s -ltermcap

# link to the graphics support libraries
LIBS = /n/elsie/work/zyda/rdobj3/lib/libreadobject.a \
/n/elsie/work/zyda/imagesupport/libnpsimage.a \
/usr/lib/libimage.a

# link to the original source code
CLIPSLIB_O = /usr/local/clips/*.o
CLIPSLIB_H = /usr/local/clips/*.h

# the executable clips code used to run the ships
ALL = gr_clips

OBJS = clips_main.o ship.o
all: $(ALL)

clean:
        rm -f *.o

delete:
        rm -f *.o $(ALL)

gr_clips: $(OBJS) $(CLIPSLIB_O) $(CLIPSLIB_H)
        cc $(GR_FLAGS) -o gr_clips $(OBJS) $(CLIPSLIB_O) \
        $(LIBS) $(MORE_LIBS)

# includes any external function definitions for clips to use
clips_main.o:clips_main.c
        cc -c clips_main.c

# compiles any changes make in the graphics code
ship.o:  ship.c variables.h
        cc -c ship.c $(LIBS) -lgl_s -lm -s $(GR_FLAGS)
```

# APPENDIX B CLIPS RULES FOR SAMPLE PROGRAM

```
; Sample graphics program driven by CLIPS. The purpose of this
; program is to pass positional data to CLIPS on the cruiser and carrier.
; CLIPS then calls the calculate_true_bearing function and orders a new heading
; for the cruiser with the true bearing to the carrier plus 30 degrees.
; The carrier is ordered to stay on the same course and speed.

;*****************************************
;
;            CLIPS RULES
;
;*****************************************

; calls C routine to start graphics code.
(defrule init-ship
        ?flag <- (init-fact)  ;starting fact, asserted to begin the process
=>
        (retract ?flag)
        (initialize_ship);call to initialize_ship, a C function
        (assert (start-fact));assert fact that starts the next rule
)


; calls ship, the main graphics driving program
; this is an "infinite loop", because it continuously calls itself
(defrule call-ship-pix
        ?flag <- (start-fact)
=>
        (retract ?flag)
        (ship)                    ; call my ship display function
        (assert (start-fact))    ; endless loop fact
        (assert (cg-fact))        ; puts cg in the world
        (assert (cv-fact))        ; puts cv in the world
)

; receives cruiser's position (x,z), heading and speed
;---------------- CG ------------------------------
 (defrule get-cg-info  " gets cruiser info "
        ?flag <- (cg-fact)
=>
        (retract ?flag)
```

```
        (bind ?cg-coord (mv-append cruiser (cruiser_values)))
        ;fact looks like (cruiser x z hd spd)
        (assert (?cg-coord))

; carrier is done in the same manner


(defrule determine-heading
        ?flag1 <- (cruiser ?cg-x ?cg-z ?cg-hd ?cg-spd)
        ?flag2 <- (carrier ?cv-x ?cv-z ?cv-hd ?cv-spd)
=>
        (retract ?flag1)
        (retract ?flag2)

        (bind ?call-tb (mv-append ?cg-x ?cg-z ?cv-x ?cv-z))
        ; tb is the true brg from cruiser to carrier
        ; calculation is done in C, with the returned value bound to ?tb
        (bind ?tb (calculate_true_bearing ?call-tb))

        ; turn cruiser to tb + 30 with the same speed
        (bind ?turn-cg (mv-append cruiser  (+ ?tb 30.0)  ?cg-spd))
        (change_cg_heading ?turn-cg)

        ; carrier maintain course and speed
        (bind ?turn-cv (mv-append carrier  ?cv-hd ?cv-spd))
        (change_cg_heading ?turn-cv)

)


;*****************************************
;
;       FACTS
;
;*****************************************

(deffacts start
        (init-fact)     ; insert a starting point
)
```

# APPENDIX C GRAPHICS CODE FOR SAMPLE PROGRAM

```
/*
Sample graphics program driven by CLIPS. The purpose of this
program is to pass positional data to CLIPS on the cruiser and carrier.
CLIPS then calls the calculate_true_bearing function and orders a heading
for the cruiser with the true bearing to the carrier plus 30 degrees.
The carrier is ordered to stay on the same course and speed.
*/

#include <stdio.h>
#include <gl.h>
#include <device.h>
#include <math.h>
#include <string.h>
#include "variables.h"  /* common definitions and constants */
#include "/usr/local/clips/clips.h"
#include <sys/times.h>
#include <sys/time.h>

/* ------------------------------------------------------ */
/* this is the C function which CLIPS calls */
void initialize_ship()
{

 /* cruiser */
 vessel[0].x_coord  = 1200.0;
 vessel[0].y_coord  = SEA_LEVEL;      /* gives initial position of cruiser */
 vessel[0].z_coord  = -400.0;
 vessel[0].roll     = 0.0;
 vessel[0].heading  = 0.0;
 vessel[0].pitch    = 0.0;
 vessel[0].speed    = 20.0;
 vessel[0].rudder   = 0.0;
 vessel[0].r        = 0.0;          /* initial angular velocity */
 vessel[0].hd_a     = -0.1;          /* related to ship's responsiveness */
 vessel[0].hd_b     = -0.05;         * related to rudder size/strength */
 vessel[0].hd_damping_var = 5.0;
 vessel[0].ordered_heading = vessel[0].heading;
 vessel[0].ordered_speed = vessel[0].speed + 1.0;


 /* carrier  same set up as cruiser */
```

```
 /* initialize the graphics */
 initialize();

 /* get the NPSOFF objects */
 vessel[0].vessel_type  = read_object("off_files/CG52.off");
 vessel[1].vessel_type  = read_object("off_files/carrier.off");
 lightobj    = read_object("off_files/the_light.off");

 /* ready the objects for display */
 ready_object_for_display(vessel[0].vessel_type);  /* cruiser */

/* make the popup menus */
 mainmenu = makethemenus();

 /* get system time */
 last_time = get_sys_time();

} /* initialize_ship */
/* ------------------------------------------------------ */

/* ------------------------------------------------------ */
void ship()
{
 /* remove for CLIPS driven graphics
      while(TRUE) {          the do forever loop */

 /* process event queue */
 while(qtest())
  {
   ...
  }

 /* calls all drawing routines */
 draw_main();

} /* end of the main ship procedure */
/* ------------------------------------------------------ */

/*********************************************************
 *
 *  begin all function definitions
 *
```

```
*************************************************************/

/*************************************************************
 *
 *        initialize()
 *
 *        - set up the iris
 *
 *************************************************************/

/*************************************************************
 *
 *        makethemenus()
 *
 *        -this routine performs all the menu construction calls
 *
 *************************************************************/

/*************************************************
 *
 *    draw_main()
 *
 *************************************************/
draw_main()
{
 winset(Main_win);
 zbuffer(TRUE);
 /* czclear sets color bitplanes in area of viewport to cval which takes packed
integer of format 0xaabbggrr, where aa is alpha, bb is blue, gg is green, rr is red */
 /* nice looking blue background */
 czclear(0xFFd42800,getgdesc(GD_ZMAX));
 loadunit(); /* must do this in Mviewing*/

 /*build the viewing matrix*/
 perspective(perspective_var,aspect_var,NEARCLIPPING,FARCLIPPING);

 /*XYZ from, XYZ to, twist*/
 lookat(eye_x, eye_y, eye_z, ref_x, ref_y,ref_z, twist);

 /* get system time from Unix */
 current_time = get_sys_time();
 delta_time = (float) (current_time - last_time);
 last_time = current_time;
```

```
 display_this_object(lightobj);
 draw_aegis();
 draw_carrier();

 /* change the buffers ... */
 swapbuffers();
}



/************************************************
 *
 *        draw_aegis()
 *
 * renders the model on the screen
 *
 ************************************************/
draw_aegis()
{
 /* these are the functions from the physically based modeling chapter */
 compute_dynamic_speed(&vessel[0].speed, &vessel[0].ordered_speed);
 turn_to_ordered_heading(&vessel[0].r,
            &vessel[0].heading,
            &vessel[0].x_coord,
            &vessel[0].z_coord,
            &vessel[0].speed,
            &vessel[0].rudder,
            &vessel[0].hd_a,
            &vessel[0].hd_b,
            &vessel[0].hd_damping_var,
            &vessel[0].ordered_heading);

 pushmatrix();
  translate(vessel[0].x_coord, vessel[0].y_coord, vessel[0].z_coord);
  rot(360.0 - vessel[0].heading, 'y');
  display_this_object(vessel[0].vessel_type);
 popmatrix();  /* main body of cruiser */

} /* draw_aegis */

 /*************************************************************
 *
```

```
 *        compute_true_bearing()
 *
 *  function to compute the true bearing from ship1 (x1,z1) to
 *  ship2 (x2,z2)
 *
 ****************************************************************/
int compute_true_bearing(int x1, int z1, int x2, int z2)
{
   float theta;          /* angle btwn points */
   int tb;               /* return variable for true bearing */


      theta = fatan2 ( (float)(z2 - z1), (float) (x2 - x1))
                                    * RAD_TO_DEGREES;
          tb = (int)theta;
      if (tb >= 360)  tb = tb - 360;
      if (tb < 0)     tb = tb + 360;

return (tb);

} /* end of compute_true_bearing */



/*************************************************************
 *
 *          CLIPS INTERFACE ROUTINES
 *
 *************************************************************/

/***********************************************
 *
 *        cruiser_values
 *
 * cruiser_values returns from the graphics program
 * to CLIPS the value of the cruiser's  X and Z
 * coordinates plus heading and speed.
 *
 ***********************************************/
VOID cruiser_values(returnValuePtr)
DATA_OBJECT_PTR returnValuePtr;
{
 VOID *multifieldPtr;
```

```
/****************************************
 *                                      *
 * check for exactly zero arguments     *
 *                                      *
 ****************************************/

if (ArgCountCheck("cruiser_values",EXACTLY,0) == -1)
{
 SetMultifieldErrorValue(returnValuePtr);
 return;
}

/************************************************
 *                                              *
 * create a multi-field value of length 4       *
 *                                              *
 ************************************************/

multifieldPtr = CreateMultifield(4);

SetMFType(multifieldPtr, 1,FLOAT);
SetMFValue(multifieldPtr,1,AddDouble(vessel[0].x_coord)); /* cruiser */

SetMFType(multifieldPtr, 2,FLOAT);
SetMFValue(multifieldPtr,2,AddDouble(vessel[0].z_coord));

SetMFType(multifieldPtr, 3,FLOAT);
SetMFValue(multifieldPtr,3,AddDouble(vessel[0].heading));

SetMFType(multifieldPtr, 4,FLOAT);
SetMFValue(multifieldPtr,4,AddDouble(vessel[0].speed));


/****************************************
 *                                      *
 * assign the type and value to the     *
 * return DATA_OBJECT                   *
 *                                      *
 ****************************************/

SetpType(returnValuePtr,MULTIFIELD);
```

```
 SetpValue(returnValuePtr,multifieldPtr);

 /****************************************
  *                                      *
  * set the begin and end points for the *
  * multifield value                     *
  *                                      *
  ****************************************/

 SetpDOBegin(returnValuePtr,1);
 SetpDOEnd(returnValuePtr,4);

 return;

}    /* end of cruiser_values */



/************************************************
 *
 *          change_ship_heading()
 *
 * receives orders from CLIPS to change the ship's
 * heading and/or speed. This function works for
 * for both the carrier and the cruiser
 *
 ************************************************/
VOID change_ship_heading()
{
 DATA_OBJECT argument;
 VOID  *multifieldPtr;
 float ordered_hd;
 char  *in_ship;
 char  *ship_str[8], *cg_str[8],  *cv_str[8];
 strcpy(cg_str, "cruiser");
 strcpy(cv_str, "carrier");

 /*************************************
  *                                   *
  * Check for exactly one argument    *
  *                                   *
  *************************************/
```

```
if (ArgCountCheck("change_ship_heading",EXACTLY,1) == -1)
 return(-1.0);

 /*************************************************
 *                                               *
 * Check that the first argument is a mv value *
 *                                               *
  ************************************************/

if (ArgTypeCheck("change_ship_heading",1,
MULTIFIELD,&argument) == 0)
 return(0L);

/*************************************************
 *                                               *
 * Get the value for the new_ships_heading   *
 * and assign its value to the global        *
 * variable, ships_heading                   *
 *                                               *
 ************************************************/

multifieldPtr = GetValue(argument);

/* ship type from CLIPS */
in_ship = ValueToString(GetMFValue(multifieldPtr,1));
strcpy(ship_str, in_ship);

/* ordered heading from CLIPS */
ordered_hd = ValueToDouble(GetMFValue(multifieldPtr,2));
if (ordered_hd < 360.0) ordered_hd += 360.0;
if (ordered_hd > 360.0) ordered_hd -= 360.0;

if ( !strcmp (cg_str, ship_str) )  /* if input ship is the cruiser */
 {
  /* ordered cg heading */
  vessel[0].ordered_heading = ordered_hd;
  /* ordered cg speed */
  vessel[0].ordered_speed = ValueToDouble(GetMFValue(multifieldPtr,3));
 }

if ( !strcmp (cv_str, ship_str) )  /* if input ship is the cv */
 {
```

```
    /* change aor values */
    vessel[1].ordered_heading = ordered_hd;
    vessel[1].ordered_speed = ValueToDouble(GetMFValue(multifieldPtr,3));
   }

 return;
}

/************************************************
 *
 *         calculate_true_bearing
 *
 * calculate_true_bearing from first vessel to the second
 * returns a single value to CLIPS
 *
 ************************************************/
double calculate_true_bearing()
{
 DATA_OBJECT argument;
 VOID *multifieldPtr;
 double x1,z1,x2,z2;
 int tb;


 /*************************************
  *                           *
  * Check for exactly one argument  *
  *                           *
  *************************************/

 if (ArgCountCheck("calculate_true_bearing",EXACTLY,1) == -1)
  return(-1.0);

 /************************************************
  *                                  *
  * Check that the first argument is a mv value *
  *                                  *
  ************************************************/

 if (ArgTypeCheck("calculate_true_bearing",1,
 MULTIFIELD,&argument) == 0)
  return(0L);
```

```
/*************************************************
 *                                               *
 * Get the value for the new_ships_heading       *
 * and assign its value to the global            *
 * variable, ships_heading                       *
 *                                               *
 *************************************************/

multifieldPtr = GetValue(argument);

/* get two pairs of x,z values from clips */
x1 = ValueToDouble(GetMFValue(multifieldPtr,1));
z1 = ValueToDouble(GetMFValue(multifieldPtr,2));
x2 = ValueToDouble(GetMFValue(multifieldPtr,3));
z2 = ValueToDouble(GetMFValue(multifieldPtr,4));

 /* call to predefined C function */
 /* this function compute_true_bearing can be used by either *
 /* the C program or called from CLIPS */
 tb = compute_true_bearing( (int)x1, (int)z1, (int)x2, (int)z2);

 if (tb >= 360)  tb = tb - 360;
 if (tb < 0)     tb = tb + 360;

 return ((double) tb);
}
```

# LIST OF REFERENCES


[Arvo91] Arvo, James, *Graphics Gems II*, Academic Press, Inc., Boston, 1991.

[Barz92] Barzel, Ronan, *Physically Based Modeling for Computer Graphics, A Structured Approach*, Academic Press, Inc., San Diego, CA, 1992.

[Bonn88] Bonnet, A., Haton, J. P., Troung-Ngoc, J. M., *Expert Systems, Principles and Practice*, Prentice Hall International (UK) Ltd., London, England, 1988.

[Cook92] Cooke, Joseph M., "*NPSNET: Flight Simulation Dynamic Modeling Using Quaternions*", M.S. Thesis, Naval Postgraduate School, Monterey, California, March 1992.

[SWOS85] Department of the Navy, Surface Warfare Officer School, *Maneuvering Board Guide*, Surface Warfare Officer Basic, Naval Education and Training Command, Newport, RI, 1985.

[USCG83] Department of Transportation, United States Coast Guard, *Navigation Rules, International-Inland*, Government Printing Office, Washington, DC, 1983.

[Fole87] Foley, James D., van Dam, Andries, Feiner, Steven K., Hughes, John F., *Computer Graphics Principles and Practice,* Addison-Wesley Publishing Company, Reading, MA, 1987.

[Giar91] Giarratano, Joseph C., *CLIPS User's Guide, Vols 1 & 2,* Software Technology Branch, NASA - Lyndon B. Johnson Space Center, Houston, TX, January 1991.

[Hall88] Halliday, David, Resnick, Robert, *Fundamentals of Physics, Third Edition Extended*, John Wiley & Sons, Inc., New York, NY, 1988.

[Hayn93] Haynes, Keith, "*Spacecraft Dynamics, 3D Visualizer*", M.S. Thesis, Naval Postgraduate School, Monterey, California, December 1993.

[Hopp92] Hoppe, William C., "*Cognitive Modeling and the Evolution of the Student Model in Intelligent Tutoring Systems,*" M.S. Thesis, Naval Postgraduate School, Monterey, California, September 1992.

[Lock] Locke, John, Pratt, David R., and Zyda, Michael J., *A DIS Network Library for UNIX and NPSNET*, Naval Postgraduate School, Monterey, CA, undated.

[Park92] Park, Hyun K., "*NPSNET: Real-Time 3D Ground-Based Vehicle Dynamics*", M.S. Thesis, Naval Postgraduate School, Monterey, California, March 1992.

[PNA88] *Principles of Naval Architecture, Volume III*, The Society of Naval Architects and Marine Engineering, 1988.

[Rowe88] Rowe, Neil C., *Artificial Intelligence Through Prolog*, Prentice Hall, Inc., Englewood Cliffs, NJ, 1988.

[Schm93] Schmidt, Dennis A., *"NPSNET: A Graphical Based Expert System To Model P-3 Aircraft Interaction With Submarines And Ships"*, M.S. Thesis, Naval Postgraduate School, Monterey, California, June 1993.

[Sieg86] Siegel, Paul., *Expert Systems: A Non-Programmer's Guide To Development and Applications*, TAB Professional and Reference Books, Blue Ridge Summit, PA, 1986.

[SGI91] Silicon Graphics, Inc., "Graphics Library Programming Guide", Document Number 007-1210-040, Mountain View, CA, 1991.

[NASA91] Software Technology Branch, *CLIPS Reference Manual, Vols I - III*, NASA - Lyndon B. Johnson Space Center, Houston, TX, January 1991.

[Swok79] Swokowski, Earl W., *Calculus with Analytic Geometry, Second Edition*, Prindle, Weber & Schmidt, Boston, MA, 1979.

[Walk90] Walker, Terri C., Miller, Richard K., *Expert Systems Handbook*, The Fairmont Press, Inc., Lilburn, GA, 1990.

[Watt92] Watt, Alan, Watt, Mark, *Advanced Animation and Rendering Techniques, Theory and Practice*, Addison-Wesley Publishing Company, New York, NY, 1992.

[Wils92] Wilson, Kalin P., Zyda, Michael J., and Pratt, David R., "NPSGDL: An Object Oriented Graphics Description Language for Virtual World Application Support", *Proceedings of the Third Eurographics Workshop on Object-Oriented Graphics, Champery, Switzerland*, 28-30 October 1992.

[Zehn93] Zehner, Stanley N., *"Modeling And Simulation Of A Deep Submergence Rescue Vehicle (DSRV) And Its Networked Application"*, M.S. Thesis, Naval Postgraduate School, Monterey, California, June 1993.

[Zyda92] Zyda, Michael J., Pratt, David R., Monahan, James D., and Wilson, Kalin P., "NPSNET: Constructing a 3D Virtual World.", *1992 Symposium on Interactive 3D Graphics*, 30 March, 1992, pp. 147-156.

[Zyda93] Zyda, Michael J., Wilson, Kalin P., Pratt, David R., Monahan, James G. and Falby, John S. "NPSOFF: An object Description Language for Supporting Virtual World Construction", *Computers & Graphics*, accepted for Vol. 17, No. 4, 1993.

# INITIAL DISTRIBUTION LIST

| | | |
|---|---|---|
| 1. | Defense Technical Information Center<br>Cameron Station<br>Alexandria, VA 22304-6145 | 2 |
| 2. | Dudley Knox Library<br>Code 52<br>Naval Postgraduate School<br>Monterey, CA 93943-5002 | 2 |
| 3. | Dr. Ted G. Lewis<br>Code CS/Lt<br>Computer Science Department<br>Naval Postgraduate School<br>Monterey, CA 93943-5000 | 1 |
| 4. | Dr. Robert B. McGhee<br>Code CS/Mz<br>Computer Science Department<br>Naval Postgraduate School<br>Monterey, CA 93943-5000 | 1 |
| 5. | Dr. David R. Pratt<br>Code CS/Pr<br>Computer Science Department<br>Naval Postgraduate School<br>Monterey, CA 93943-5000 | 4 |
| 6. | Dr. Sehung Kwak<br>Code CS/Kw<br>Computer Science Department<br>Naval Postgraduate School<br>Monterey, CA 93943-5000 | 2 |
| 7. | Dr. Hemant Bhargava<br>Code AS/Bh<br>Administrative Science Department<br>Naval Postgraduate School<br>Monterey, CA 93943-5000 | 1 |
| 8. | LT John H. Hearne<br>318 Mill Street<br>Worcester, MA 01602 | 2 |